FTP_Algorithms

Cheat Sheet

Diego Gil

Herbstsemester 2024/25

Contents

1 Data Structures

1.1 Trees

1.1.1 Basic Tree Terminology

Height: The height of a tree is the length of the longest path from the root to a leaf. It is the number of edges on this path.

Level: The level of a node is the number of edges on the path from the root to the node. The root node is at level 0.

Minimum Width: The minimum width of a tree is the smallest number of nodes at any level of the tree.

Maximum Width: The maximum width of a tree is the largest number of nodes at any level of the tree.

Depth: The depth of a node is the number of edges from the node to the tree's root node.

Leaf: A leaf is a node with no children.

Internal Node: An internal node is a node with at least one child.

Binary Tree: A tree data structure in which each node has at most two children, referred to as the left child and the right child.

1.1.2 KD-Trees

Problem Type: Construction of a KD-Tree from 2D points

What to Look For:

- Set of 2D points given as coordinates
- Request to build a KD-Tree
- Questions about tree properties (height, leaves)

Given Points: $P = \{(1,3), (12,1), (4,5), (5,4), (10,11), (8,2), (2,7)\}$

Solution Strategy:

1. Sort points by x-coordinate (root level)
2. Find median point
3. Split into left/right subtrees
4. Repeat with y-coordinates for next level
5. Continue alternating x/y until all points placed

Detailed Solution:

1. **Root Level (x-split)**
   - Sorted x: $(1,3), (2,7), (4,5), (\mathbf{5,4}), (8,2), (10,11), (12,1)$
   - Median $(5,4)$ becomes root ℓ_1

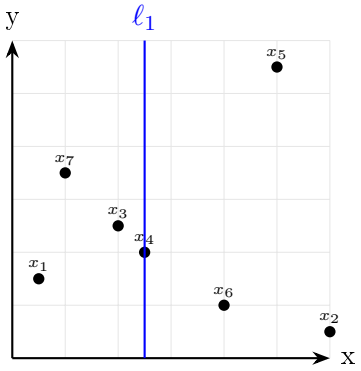


Figure 1: *
Coordinate Split at Root Level

2. **Tree Structure**

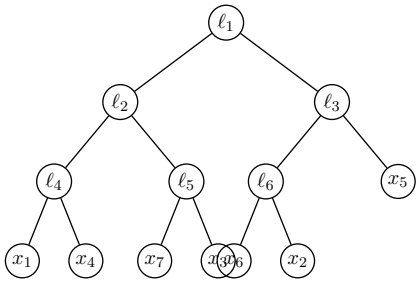


Figure 2: *
KD-Tree Structure

3. **Final Properties**
   - Height: 3 (counting from 0)
   - Leaves: 7 (all original points)
   - Second leaf from left: $(4,5)$

Exam Tips:

1. Always start by sorting points on current dimension
2. Mark median point clearly in your sorting
3. Draw coordinate system with splitting lines
4. Keep track of which dimension you're splitting on:
   - Level 0: x-coordinate
   - Level 1: y-coordinate
   - Level 2: x-coordinate
   - And so on...
5. Verify tree properties at the end

Common Mistakes to Avoid:

- Don't forget to alternate dimensions
- Don't skip sorting at each level
- Don't mix up left ($<$) and right ($>$) subtrees
- Don't forget to verify final tree properties

1.1.3 KD-Tree Complexity Analysis

Problem Type: Complexity proof for KD-Tree construction

What to Look For:

- Proof of time complexity $O(n \log n)$

- Proof of space complexity $O(n)$
- Recursive analysis

Solution Strategy:

1. Prove space complexity first (easier)
2. Analyze recursive structure
3. Set up recurrence relation
4. Apply Master Theorem

Space Complexity Proof:

1. For $n = 2^k$ points:
   - Internal nodes (parents): $2^k - 1$
   - Total nodes: $2^k + 2^{k-1} = n + n/2 = 3n/2 < 3n$
2. For general n (not power of 2):
   - Find t where $2^{t-1} < n < 2^t$
   - Internal nodes n_p: $2^{t-2} < n_p < 2^{t-1}$
   - Total nodes: $3 \cdot 2^{t-2} < n + n_p < 3 \cdot 2^{t-1}$
   - Therefore: $n + n_p < 3n$
3. Each node uses $O(1)$ storage
4. Total storage: $O(1) \cdot O(n) = O(n)$

Time Complexity Proof:

1. At each recursion:
   - Split n points into two subsets of $n/2$
   - Finding median costs $O(n)$
2. Recurrence relation:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n > 1 \end{cases}$$

3. Apply Master Theorem:
   - Similar to Merge-Sort analysis
   - Results in $T(n) = O(n \log n)$

Key Points for Exam:

- Space complexity proof:
  - Count nodes for power of 2
  - Extend to general case
  - Multiply by constant storage
- Time complexity proof:
  - Identify recursive pattern
  - Write recurrence relation
  - Apply Master Theorem
- Remember median finding is $O(n)$

Common Mistakes to Avoid:

- Don't forget to account for non-power-of-2 cases
- Don't ignore constant factors in space analysis
- Remember to justify linear median finding
- Don't skip the Master Theorem application

1.1.4 KD-Tree Implementation Details

Definition: A k-dimensional tree (KD-tree) is a space-partitioning data structure for organizing points in a k-dimensional space.

Properties:

- Each node represents a point in k-dimensional space
- Each non-leaf node splits space into two parts
- Splits alternate between dimensions at each level
- Points are stored in a binary tree structure

BUILDKDTREE Algorithm:

1. Input: Set of points P and depth d
2. If $|P| = 1$: return leaf node with the point
3. Determine split dimension: $d \bmod k$ (k=2 for 2D)
4. Sort points by split dimension
5. Find median point m
6. Split points into P_L ($< m$) and P_R ($> m$)
7. Recursively build left and right subtrees:
   - $T_L = \text{BUILDKDTREE}(P_L, d+1)$
   - $T_R = \text{BUILDKDTREE}(P_R, d+1)$
8. Return node with (m, T_L, T_R)

Implementation Details:

- Even depths: split on x-coordinate (vertical line)
- Odd depths: split on y-coordinate (horizontal line)
- Each internal node stores:
  - Split point coordinates
  - Split dimension
  - Pointers to left/right children
- Leaf nodes store single points

1.1.5 Binary Search Trees (BST)

Definition: A binary tree where for each node x:

- All keys in left subtree are $< x.key$
- All keys in right subtree are $> x.key$
- No duplicate keys allowed

Basic Operations:

1. **TREE-SEARCH**(x, k): Find node with key k
   - Start at root, compare with k
   - If equal: found
   - If k smaller: go left
   - If k larger: go right
   - Time: $O(h)$ where h is height
2. **TREE-MINIMUM**(x): Find smallest key
   - Follow left pointers until NIL
   - Time: $O(h)$
3. **TREE-MAXIMUM**(x): Find largest key
   - Follow right pointers until NIL
   - Time: $O(h)$
4. **TREE-SUCCESSOR**(x): Find next larger key
   - If right subtree exists: TREE-MINIMUM(right)
   - Else: Go up until first right turn
   - Time: $O(h)$
5. **TREE-PREDECESSOR**(x): Find next smaller key
   - If left subtree exists: TREE-MAXIMUM(left)
   - Else: Go up until first left turn
   - Time: $O(h)$

Modifying Operations:

1. **TREE-INSERT**(T, z): Insert new node z
   - Follow BST property down to leaf
   - Insert as left/right child
   - Time: $O(h)$
2. **TREE-DELETE**(T, z): Delete node z
   - Case 1: No children - remove directly
   - Case 2: One child - replace with child
   - Case 3: Two children:
     - Find successor y (min in right subtree)
     - Replace z with y
     - Delete y from original position
   - Time: $O(h)$

Helper Operation:

- **TRANSPLANT**(T, u, v): Replace subtree
  - Replaces subtree rooted at u with subtree rooted at v
  - Updates parent pointers
  - Used in DELETE operation

Properties:

- Inorder traversal gives sorted sequence
- Height h determines operation time:
  - Best case (balanced): $h = \lg n$
  - Worst case (linear): $h = n$
- No explicit balancing - shape depends on insertion order

Tree Traversal:

- **Inorder**: Left subtree → Root → Right subtree
  - Visits nodes in sorted order
  - Used for ordered printing
- **Preorder**: Root → Left subtree → Right subtree
  - Root processed before children
  - Used for copying tree structure
- **Postorder**: Left subtree → Right subtree → Root
  - Root processed after children
  - Used for deletion

Implementation Details:

- Node structure:
  - key: Value stored in node
  - left, right: Pointers to children
  - p: Pointer to parent (optional)
- Sentinel NIL:
  - Used to mark leaf nodes
  - Simplifies boundary conditions

Key Insights:

- Successor never has left child
- Predecessor never has right child
- All operations maintain BST property
- Performance depends on tree height
- Balancing requires additional mechanisms (AVL, Red-Black)

2 Graph Theory

2.1 Basic Definitions

Graph: A graph $G = (V, E)$ consists of:

- V: Set of vertices (nodes)
- E: Set of edges connecting vertices
- For directed graphs: $E \subseteq V \times V$
- For undirected graphs: E contains unordered pairs

Graph Operations:

- **Transpose:** $G^T = (V, E^T)$ where $E^T = \{(u, v) \mid (v, u) \in E\}$
- **Subgraph:** Graph $H = (W, F)$ where $W \subseteq V$ and $F \subseteq E$
- **Path:** Sequence of vertices connected by edges
- **Cycle:** Path that starts and ends at same vertex

Graph Properties:

- **Connected:** Path exists between any two vertices
- **Strongly Connected:** Directed path exists between any two vertices
- **Tree:** Connected graph with no cycles
- **DAG:** Directed Acyclic Graph
- **Dense:** $|E| \approx |V|^2$
- **Sparse:** $|E| \ll |V|^2$

Graph Representations:

1. **Adjacency Matrix:**
   - $n \times n$ matrix where $n = |V|$
   - Entry m_{ij} is 1 if edge (i, j) exists, 0 otherwise
   - For weighted graphs: m_{ij} contains edge weight
   - Space complexity: $\Theta(|V|^2)$
   - Good for dense graphs
   - Operations:
     - Edge lookup: $O(1)$
     - Add/remove edge: $O(1)$
     - Add vertex: $O(|V|^2)$
     - Find neighbors: $O(|V|)$
     - Transpose: $O(|V|^2)$
2. **Adjacency List:**
   - Array of $|V|$ linked lists
   - For each vertex, store list of adjacent vertices
   - Space complexity: $\Theta(|V| + |E|)$
   - Better for sparse graphs
   - Operations:
     - Edge lookup: $O(\text{degree}(v))$
     - Add edge: $O(1)$
     - Remove edge: $O(\text{degree}(v))$
     - Add vertex: $O(1)$
     - Find neighbors: $O(1)$
     - Transpose: $O(|V| + |E|)$

2.1.1 Shortest Paths

Single-Source Shortest Path Problem:

- Find shortest paths from source s to all vertices
- Path weight: Sum of edge weights along path
- Different algorithms for different scenarios:
  - Dijkstra: Non-negative weights
  - Bellman-Ford: General weights
  - BFS: Unweighted graphs

Path Properties:

- **Optimal Substructure:** Subpaths of shortest paths are shortest paths
- **Triangle Inequality:** $\delta(s, v) \leq \delta(s, u) + w(u, v)$
- **No Negative Cycles:** Required for well-defined shortest paths
- **Upper Bound Property:** $v.d \geq \delta(s, v)$ always

2.1.2 Dijkstra's Algorithm

Purpose: Find shortest paths from source vertex to all other vertices

Requirements:

- Non-negative edge weights
- Can be directed or undirected graph

Algorithm Steps:

1. Initialize:
   - $s.d = 0$ (source distance)
   - $v.d = \infty$ for all other vertices
   - $Q = V$ (priority queue)
   - $v.\pi = \text{NIL}$ for all vertices (predecessors)
2. While Q not empty:
   - $u = \text{Extract-Min}(Q)$
   - For each edge (u, v):
     - Relax: If $v.d > u.d + w(u, v)$:
       - $v.d = u.d + w(u, v)$
       - $v.\pi = u$ (predecessor)

Correctness:

- Invariants:
  - $v.d \geq \delta(s, v)$ for all $v \in V$
  - If $v.\pi \neq \text{NIL}$, then $v.d = v.\pi.d + w(v.\pi, v)$
  - Once v extracted from Q, $v.d = \delta(s, v)$
- Fails with negative weights because:
  - Assumes adding edge cannot decrease path weight
  - Negative cycles can create arbitrarily small paths
  - Once vertex extracted, its distance assumed final

Implementation Details:

- Priority Queue Options:

  - Binary Heap: $O((|V| + |E|) \log |V|)$
  - Fibonacci Heap: $O(|V| \log |V| + |E|)$
  - Array: $O(|V|^2)$
- Space Complexity: $O(|V|)$
- Path Reconstruction: Follow π pointers from target to source

3 Complexity Analysis

3.1 Sorting Complexity

Big-Oh Notation: Describes the upper bound of an algorithm's running time. For example, $O(n \log n)$ is common in efficient sorting algorithms like Merge Sort.

3.2 Quadratic Algorithms

Understanding $O(n^2)$: Often seen in simple sorting algorithms like Bubble Sort, where each element is compared to every other element.

3.3 Time Complexity

Complexity Classes: Includes constant $O(1)$, logarithmic $O(\log n)$, linear $O(n)$, quadratic $O(n^2)$, and more. Helps in understanding the efficiency of algorithms.

3.4 Dominant Terms

Identifying Dominant Terms: In expressions like $5n^2 + 3n \log n$, the term $5n^2$ is dominant, leading to $O(n^2)$.

3.5 Big-Oh Notation Properties

Rules: Includes the rule of sums $O(f + g) = O(\max\{f, g\})$ and products $O(f \cdot g) = O(f) \cdot O(g)$.

3.6 Computational Complexity

Nested Loops: Analyzing loops within loops to determine total complexity, such as $O(n(\log n)^2)$ for certain nested structures.

3.7 Master Theorem

The Master Theorem provides a way to solve recurrence relations of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where $a \geq 1$, $b > 1$, and $f(n)$ is an asymptotically positive function. The theorem helps determine the asymptotic behavior of $T(n)$ by comparing $f(n)$ with $n^{\log_b a}$:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then:

$$T(n) = \Theta(n^{\log_b a})$$

2. If $f(n) = \Theta(n^{\log_b a})$, then:

$$T(n) = \Theta(n^{\log_b a} \log n)$$

3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and sufficiently large n, then:

$$T(n) = \Theta(f(n))$$

The Master Theorem is widely used in analyzing the time complexity of divide-and-conquer algorithms, such as Merge Sort and Quick Sort.

3.8 Heap Operations

Basic Heap Properties:

- A heap is a complete binary tree
- In a max-heap, for each node i: parent.key $\geq$ children.key
- In a min-heap, for each node i: parent.key $\leq$ children.key

Array Representation: For a node at index i:

- Parent: $\lfloor i/2 \rfloor$
- Left child: $2i$
- Right child: $2i + 1$

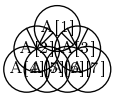


Figure 3: *
Array indices in heap

MAX-HEAPIFY Operation:

1. Compare root with children
2. If child is larger, swap with largest child
3. Recursively heapify affected subtree

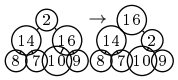


Figure 4: *
MAX-HEAPIFY example

BUILD-MAX-HEAP Operation:

1. Start from last non-leaf node ($\lfloor n/2 \rfloor$)
2. Apply MAX-HEAPIFY to each node up to root

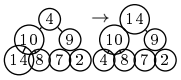


Figure 5: *
BUILD-MAX-HEAP example

HEAPSORT Operation:

1. BUILD-MAX-HEAP
2. Repeatedly:
   - Swap root with last element
   - Reduce heap size by 1

- MAX-HEAPIFY root

COUNTING-SORT Overview: A non-comparison based sorting algorithm that works in $O(n+k)$ time, where n is the number of elements and k is the range of input.

Key Properties:

- Stable sorting algorithm
- Works best when $k = O(n)$
- Requires extra space for counting array C and output array B
- Input must be non-negative integers

Algorithm Steps:

1. Initialize counting array $C[0..k]$ to all zeros
2. Count occurrences of each element in input array A
3. Compute cumulative sums in C
4. Build output array B using C as position guide

Pseudocode:

```
1: function COUNTING-SORT(A, B, k)
2:     let C[0..k] be a new array
3:     for i ← 0 to k do
4:         C[i] ← 0
5:     end for
6:     for i ← 1 to A.length do
7:         C[A[i]] ← C[A[i]] + 1
8:     end for
9:     for i ← 1 to k do
10:        C[i] ← C[i] + C[i − 1]
11:    end for
12:    for j ← A.length downto 1 do
13:        B[C[A[j]]] ← A[j]
14:        C[A[j]] ← C[A[j]] − 1
15:    end for
16: end function
```

Array States During Execution:

- After counting ($C[i]$ = frequency of i):
  - Each $C[i]$ contains count of elements equal to i
- After cumulative sums:
  - Each $C[i]$ contains count of elements $\leq i$
  - $C[i]$ represents position after which next i should go
- During output array construction:
  - Process input from right to left
  - Use $C[A[j]]$ as position index in B
  - Decrement $C[A[j]]$ after each placement

Time Complexity Analysis:

- Initialize C: $O(k)$
- Count frequencies: $O(n)$
- Compute cumulative sums: $O(k)$
- Build output array: $O(n)$
- Total: $O(n+k)$

Space Complexity:

- Array C: $O(k)$
- Array B: $O(n)$
- Total: $O(n+k)$

Key Insights:

- Processing from right to left ensures stability
- Cumulative sum array C determines final positions
- No comparisons between elements needed
- Efficient when range of input is not too large

Common Applications:

- Sorting integers with known range
- As subroutine in Radix Sort
- When stability is required
- When input range is $O(n)$

3.9 Week 7: Theoretical Concepts

3.9.1 Polar Angles

Definition: The polar angle of a point p_i with respect to an origin p_0 is the angle from the semi-horizontal straight line r and the vector $\overrightarrow{p_0p_i}$. The angle is measured counterclockwise and is in the interval $[0, 2\pi)$.

Applications:

- Sorting points by polar angle for computational geometry tasks.
- Efficient algorithms with $O(n \log n)$ complexity.

3.9.2 Segment Intersections

ANY-SEGMENTS-INTERSECT Algorithm:

- Detects if any segments intersect at a point.
- Uses a sweep line approach to efficiently handle intersections.
- Handles vertical segments by treating endpoints appropriately.

Key Insights:

- The algorithm efficiently processes event points with $O(n \log n)$ complexity.
- Correctly identifies intersections involving multiple segments, including vertical ones.

4 Graph Algorithms

4.1 Graph Representations

4.1.1 Graph Transpose

Problem Type: Computing transpose G^T of a di-

What to Look For:

- Graph representation type (matrix/list)
- Direction of edges must be reversed
- Time complexity analysis required

Key Definitions:

- $G^T = (V, E^T)$ where $E^T = \{(v, u) \mid (u, v) \in E\}$
- $|V| = n$ (number of vertices)
- $|E|$ (number of edges)

Solution for Adjacency Matrix:

1. Given matrix M_G, create M_G^T by swapping entries:

$$M = \begin{pmatrix} m_{11} & m_{12} & \cdots & m_{1n} \\ m_{21} & m_{22} & \cdots & m_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{n2} & \cdots & m_{nn} \end{pmatrix}$$

$$M^T = \begin{pmatrix} m_{11} & m_{21} & \cdots & m_{n1} \\ m_{12} & m_{22} & \cdots & m_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ m_{1n} & m_{2n} & \cdots & m_{nn} \end{pmatrix}$$

2. Example:

$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, M^T = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

3. Time Complexity: $\Theta(n^2)$
   - Must swap $n^2 - n$ entries (excluding diagonal)
   - Each swap is $O(1)$

Solution for Adjacency List:

1. Create empty adjacency lists for G^T: $O(n)$
2. For each vertex v in G:
   - For each edge (v, w) in v's adjacency list
   - Add v to w's list in G^T
3. Time Complexity: $\Theta(|V| + |E|)$
   - Creating lists: $O(|V|)$
   - Processing edges: $O(|E|)$

Comparison:

- Matrix: $\Theta(n^2)$ always
- List: $\Theta(|V| + |E|)$ which is better for sparse graphs
- List requires more complex implementation

Common Mistakes to Avoid:

- Don't forget self-loops (diagonal elements)
- Don't count diagonal elements in matrix swaps
- Remember to initialize all new lists in adjacency list solution
- Don't confuse $|V|$ and $|E|$ in complexity analysis

4.2 Shortest Paths

4.2.1 Dijkstra's Algorithm Limitations

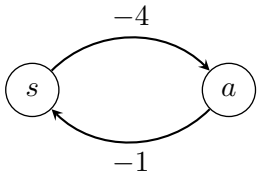
Problem Type: Counterexample for Dijkstra with negative weights

What to Look For:

- Directed graph with negative weights
- Minimal example showing algorithm failure
- Negative cycle demonstration

Solution:

1. Consider this directed graph:



2. Why Dijkstra fails:
   - Initial distance to a: -4
   - After one cycle: -5
   - After two cycles: -6
   - Continues to decrease indefinitely

Key Properties:

- Any negative cycle causes Dijkstra to fail
- Algorithm assumes:
  - Edge weights are non-negative
  - Shortest paths exist (no negative cycles)
- For negative weights, use Bellman-Ford instead

Common Mistakes to Avoid:

- Single negative edge isn't enough
- Example must have negative total cycle weight

5 Exercises Part 1

5.1 Exercise 1.1: Sorting Complexity

Problem: A sorting method with "Big-Oh" complexity $O(n \log n)$ spends exactly 1 millisecond to sort 1,000 data items. Given this, estimate how long it will take to sort 1,000,000 items.

Solution Steps:

1. Understand the Problem: You need to find out how long it will take to sort 1,000,000 items using the given complexity.
2. Identify Known Values:
   - $T(1,000) = 1ms$
   - Complexity is $O(n \log n)$
3. Calculate Constant c:
   - Formula: $T(n) = c \cdot n \log n$
   - Use $T(1,000) = 1ms$ to find c:

$$c = \frac{1ms}{1,000 \log 1,000}$$

4. Calculate $T(1,000,000)$:
   - Use the formula $T(n) = c \cdot n \log n$
   - Substitute $n = 1,000,000$:

$$T(1,000,000) = c \cdot 1,000,000 \cdot \log 1,000,000$$

5. Simplify the Expression:
   - Calculate $\log 1,000,000$
   - Multiply and simplify to find the time in seconds.

Exam Note: Remember that $O(n \log n)$ complexity means the time increases logarithmically with the size of the data.

Hint: To solve similar exercises, focus on understanding the relationship between the given complexity and the time it takes to process a certain amount of data. Use the formula $T(n) = c \cdot f(n)$ to calculate the constant c and then use it to find the time for a different amount of data.

5.2 Exercise 1.2: Quadratic Algorithm

Problem: A quadratic algorithm with processing time $T(n) = cn^2$ spends 1ms for 100 items. Calculate the time for 5,000 items.

Solution Steps:

1. Understand the Problem: You need to calculate the time for 5,000 items given the complexity.
2. Identify Known Values:
   - $T(100) = 1ms$
   - Complexity is $O(n^2)$
3. Calculate Constant c:
   - Formula: $T(n) = c \cdot n^2$
   - Use $T(100) = 1ms$ to find c:

$$c = \frac{1ms}{100^2}$$

4. Calculate $T(5,000)$:
   - Use the formula $T(n) = c \cdot n^2$
   - Substitute $n = 5,000$:

$$T(5,000) = c \cdot (5,000)^2$$

5. Simplify the Expression:
   - Calculate $(5,000)^2$
   - Multiply and simplify to find the time in milliseconds.

Exam Note: Quadratic complexity $O(n^2)$ means time increases with the square of the data size.

Hint: To solve similar exercises, focus on understanding the relationship between the given complexity and the time it takes to process a certain amount of data. Use the formula $T(n) = c \cdot f(n)$ to calculate the constant c and then use it to find the time for a different amount of data.

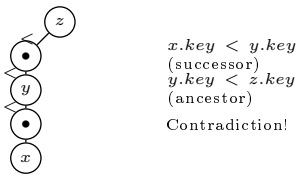


Figure 6: *
Contradiction in BST property

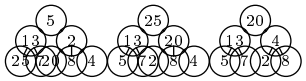


Figure 7: *
HEAPSORT steps: Initial → BUILD-MAX-HEAP → First extraction

5.3 Exercise 3.2: Tree Predecessor

Problem: Write the TREE-PREDECESSOR procedure.

Solution: To obtain TREE-PREDECESSOR(x) procedure, we replace in TREE-SUCCESSOR(x) "left" instead of "right" and "MAXIMUM" instead of "MINIMUM".

```
1: procedure TREE-PREDECESSOR(x)
2:     if x.right ≠ NIL then
3:         return Tree-Maximum(x.left)
4:     end if
5:     y ← x.p
6:     while y ≠ NIL and x = y.left do
7:         x ← y
8:         y ← y.p
9:     end while
10:    return y
11: end procedure
```

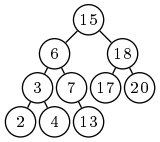


Figure 8: *
Example: Predecessor of 15 is 13 (maximum in left subtree)

Explanation:

- Case 1: If x has a left subtree, the predecessor is the maximum element in that subtree
- Case 2: If no left subtree exists, we go up the tree until we find a node that is a right child
- The predecessor's key is the largest key in the tree smaller than $x.key$

5.4 Exercise 3.4: Binary Search Tree Insertion

Problem: Let T be a Binary Search Tree. Prove that it always possible to insert a node z as a leaf of the tree T with $z.key = r$.

Solution: This is a straightforward property of Binary Search Trees. We prove this by induction on the height of the tree.

Proof.

- **Base case** ($h = 0$):
  - Tree consists only of root node x
  - If $r \leq x.key$: place z as left child of x
  - If $r > x.key$: place z as right child of x
- **Inductive step:**
  - Assume the statement is true for trees of height $h - 1$
  - For a tree of height h with root x:
    - If $r \leq x.key$: insert in left subtree
    - If $r > x.key$: insert in right subtree
  - By inductive hypothesis, we can insert in the chosen subtree (height $h - 1$)

□

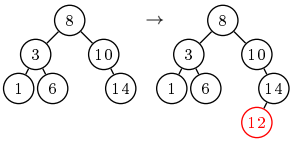


Figure 9: *
Example: Inserting node with key=12 (shown in red)

Key Points:

- The BST property ensures we can always find a valid leaf position
- At each step, we reduce the problem to a smaller subtree
- The process terminates when we reach a NULL child pointer
- Insertion maintains the BST property

5.5 Exercise 3.5: Binary Search Tree Deletion

Problem: Let T be a Binary Search Tree given in the figure below. Give the output tree after the call of TREE-DELETE(T, z) where z is the node with key 41.

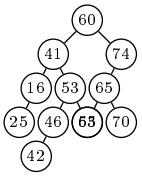


Figure 10: *
Initial Binary Search Tree with node 41 to be deleted

Algorithm: TREE-DELETE(T, z)

```
1: procedure TREE-DELETE(T, z)
2:     if z.left = NIL then
3:         TRANSPLANT(T, z, z.right)
4:     else if z.right = NIL then
5:         TRANSPLANT(T, z, z.left)
6:     else
7:         y ← Tree-Minimum(z.right)
8:         if y.p ≠ z then
9:             TRANSPLANT(T, y, y.right)
10:            y.right ← z.right
11:            y.right.p ← y
12:        end if
13:        TRANSPLANT(T, z, y)
14:        y.left ← z.left
15:        y.left.p ← y
16:    end if
17: end procedure
```

Solution: Let's solve this step by step following the TREE-DELETE algorithm:

1. **Analyze the node to be deleted (41):**

- Node 41 has two children: 16 (left) and 53 (right)
- Since it has two children, we fall into the third case (lines 7-15)
- We need to find its successor to replace it

2. **Find the successor of 41 (lines 7):**
   - Call TREE-MINIMUM($z.right$) to find successor
   - Right subtree starts at node 53
   - Follow left pointers: 53 → 46 → 42
   - Node 42 has no left child, so it's the successor
3. **Handle successor's position (lines 8-11):**
   - Check if successor (42) is not a direct child of 41
   - Since 42 is not direct child (it's grandchild), we:
     - Replace 42 with its right child (NIL in this case)
     - Make 42 point to 41's right child (53)
     - Make 53's parent point to 42
4. **Complete the replacement (lines 12-14):**
   - Replace 41 with 42 using TRANSPLANT
   - Make 42 point to 41's left child (16)
   - Make 16's parent point to 42

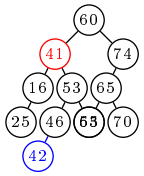


Figure 11: *
Finding successor: Node to delete (41) in red, successor (42) in blue

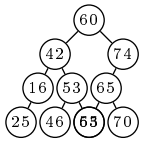


Figure 12: *
Final Binary Search Tree after deleting node 41

Key Points for Tree Deletion:

- There are three cases when deleting a node:
  1. Node has no children (leaf node):
     - Simply remove it by setting parent's pointer to NIL
     - Example: Deleting a leaf like node 25
  2. Node has one child:
     - Replace node with its only child
     - Update parent pointers
     - Example: If node 16 had only child 25
  3. Node has two children:
     - Find successor (smallest value in right subtree)
     - Replace node with successor
     - Handle successor's original position
     - Example: Node 41 in our case
- Finding the successor (TREE-MINIMUM):
  - Start at node's right child
  - Keep following left pointers until NIL
  - Last node found is successor
  - Important: Successor never has a left child
- TRANSPLANT operation:
  - Used to replace one subtree with another
  - Updates parent pointers correctly
  - Handles special case of root node
  - Does not handle child pointers of moved nodes

Verification: After deletion:

- Node 42 maintains BST property:
  - Left subtree (16, 25) contains values < 42
  - Right subtree (53, 46, 55) contains values > 42
- Tree structure remains valid:
  - All parent-child pointers are correct
  - No nodes were lost or duplicated
- BST invariants are preserved:
  - For every node: left subtree values $<$ node key $<$ right subtree values
  - Tree remains connected
  - No cycles are created

5.6 Exercise 4.1: Quicksort Partitioning Worst Case

Problem: Prove that the worst case in Partitioning Algorithm (for Quicksort) has running time $\Theta(n^2)$, where n is the cardinality of the set of elements in the partitioning.

Solution: We provide both an intuitive proof and a formal proof by induction.

Part 1: Intuitive Proof

1. **Worst Case Scenario:**
   - At each step, we get maximally unbalanced partitions:
   - A $k-1$ element array and an empty array
   - This happens when pivot is always smallest/largest element
2. **Recurrence Relation:** Let $T(n)$ be the running time of Quicksort with Partition:
   - Splitting time is linear: $\Theta(k)$ for array of size k
   - Base case: $T(0)$ is constant, so $T(0) = \Theta(1)$
   - For size k: $T(k) = T(k-1) + T(0) + \Theta(k)$
   - Simplifies to: $T(k) = T(k-1) + \Theta(k)$
3. **Solving the Recurrence:**

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) \\ &= T(n-2) + \Theta(n-1) + \Theta(n) \\ &= T(n-3) + \Theta(n-2) + \Theta(n-1) + \Theta(n) \\ &\vdots \\ &= T(0) + \sum_{i=1}^{n} \Theta(i) \end{aligned}$$

4. **Final Step:**
   - Sum is arithmetic series: $\sum_{i=1}^{n} i$
   - Using identity: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
   - Therefore: $T(n) = \Theta(n^2)$

Part 2: Formal Proof by Induction

1. **Claim:** $T(n) = \Theta(n^2)$ for worst-case running time
2. **Precise Statement:**
   - For all $0 < m < n$: $T(m) = \Theta(m^2)$
   - This means $\exists c_1, d_1 > 0 : c_1 m^2 \leq T(m) \leq d_1 m^2$
   - Partition time $P(m) = \Theta(m)$, so $\exists c_2, d_2 > 0 : c_2 m \leq P(m) + T(0) \leq d_2 m$
3. **Constants:**
   - Let $c = \min\{c_1, c_2\}$ and $d = \max\{d_1, d_2, 1\}$
   - Then for all $m \geq n-1$: $cm^2 \leq T(m) \leq dm^2$
   - And for all $m \geq n$: $2cm \leq T(0) + P(m) \leq dm$
4. **Inductive Step:**

$$T(n) = T(n-1) + T(0) + P(n)$$
$$c(n-1)^2 + 2cn \leq cn^2 - 2cn + 1 + 2cn = cn^2 + 1$$
$$d(n-1)^2 + dn \leq dn^2 - dn + 1 \leq dn^2$$

Therefore: $cn^2 < T(n) \leq dn^2$, proving $T(n) = \Theta(n^2)$

Key Insights:

- The worst case occurs with extremely unbalanced partitions
- Each partition step costs linear time
- The cumulative effect leads to quadratic runtime
- Both intuitive and formal proofs confirm $\Theta(n^2)$ complexity

5.7 Exercise 4.2: COUNTING-SORT Algorithm

Problem: We apply COUNTING-SORT with the input vector $A = (5, 6, 5, 3, 3, 7, 4, 4, 4, 5, 3, 8, 8)$. Let C and B be the arrays mentioned in the pseudocode of COUNTING-SORT. Answer the following questions:

1. What is $C[7]$ after the for loop at lines 7-8 of the pseudocode?
2. What is $B[13]$ after the first cycle at line 10?
3. What is $C[8]$ after the first cycle?

Solution: Let's solve this step by step following the COUNTING-SORT algorithm.

1. Algorithm Overview:

- Input array $A = (5, 6, 5, 3, 3, 7, 4, 4, 4, 5, 3, 8, 8)$ with $k = 8$ (max value)
- Array $C[0..k]$ is used for counting and cumulative sums
- Array $B[1..n]$ will store the sorted output

2. Step-by-Step Execution:

1. **Initialize array C:**
   - Create $C[0..8]$ initialized to zeros
   - $C = [0, 0, 0, 0, 0, 0, 0, 0, 0]$
2. **Count elements (lines 3-4):**
   - Count occurrences of each value in A
   - After this step: $C = [0, 0, 0, 3, 3, 3, 1, 1, 2]$
   - Meaning: three 3s, three 4s, three 5s, one 6, one 7, two 8s
3. **Compute cumulative sums (lines 5-6):**
   - Transform C into cumulative counts
   - $C = [0, 0, 0, 3, 6, 9, 10, 11, 13]$
   - Therefore, $C[7] = 11$ (**Answer to Question 1**)
4. **Build sorted array (lines 7-8):**
   - Process A from right to left
   - Place elements in B based on C values
   - After first cycle (processing last element 8):
     - $B[13] = 8$ (**Answer to Question 2**)
     - $C[8]$ is decremented to 12 (**Answer to Question 3**)

- Final sorted array: $B = [3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 7, 8, 8]$

Key Insights:

- The cumulative sum array C helps maintain stability by tracking positions
- Processing from right to left ensures stability
- $C[i]$ represents the position after which the next element i should be placed
- Each placement decrements the corresponding counter in C

Final Answers:

1. $C[7] = 11$
2. $B[13] = 8$
3. $C[8] = 12$

5.8 Exercise 5.1: BUILDKDTREE Algorithm

Problem: We apply BUILDKDTREE$(P, 0)$ to the following set P of points in the plane:

$$P = \{(1, 3), (12, 1), (4, 5), (5, 4), (10, 11), (8, 2), (2, 7)\}$$

Answer the following questions:

1. Give the height of the tree
2. How many leaves are there?
3. The second leaf (starting from left) is the point with first coordinate...

Solution: Let's solve this step by step following the BUILDKDTREE algorithm.

1. Algorithm Review:

- At even depths (0,2,...): split on x-coordinate (vertical line)
- At odd depths (1,3,...): split on y-coordinate (horizontal line)
- Each split creates a new level in the tree
- Points are recursively divided into left and right subsets

2. Step-by-Step Construction:

1. **Root Level (depth 0, x-split):**
   - Sort by x-coordinate: $(1, 3), (2, 7), (4, 5), (5, 4), (8, 2), (10, 11), (12, 1)$
   - Median point $(5, 4)$ becomes root ℓ_1
   - Left subset: $\{(1, 3), (2, 7), (4, 5)\}$
   - Right subset: $\{(8, 2), (10, 11), (12, 1)\}$
2. **Level 1 (depth 1, y-split):**
   - Left subset sorted by y: $(1, 3), (4, 5), (2, 7)$
   - Right subset sorted by y: $(12, 1), (8, 2), (10, 11)$
   - Medians $(2, 7)$ and $(10, 11)$ become nodes ℓ_2 and ℓ_3
3. **Level 2 (depth 2, x-split):**
   - Split remaining points into leaf nodes
   - Left of ℓ_2: $(1, 3), (4, 5)$ become leaves under ℓ_4
   - Left of ℓ_3: $(8, 2), (12, 1)$ become leaves under ℓ_6

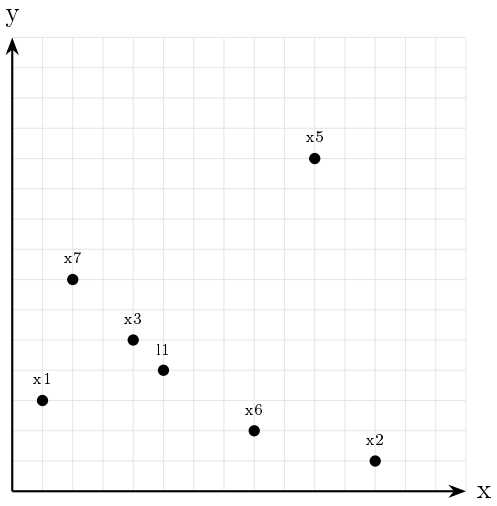


Figure 13: *
Points in 2D plane

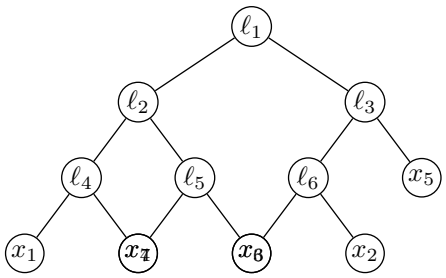


Figure 14: *
Final KD-Tree structure

3. Answers:

1. Height of the tree = 3 (counting levels from 0)
2. Number of leaves = 7 (each original point becomes a leaf)
3. Second leaf from left = $x_4 = (4, 5)$

Key Insights:

- The tree is built by alternating between x and y coordinates
- Each internal node represents a splitting line
- Vertical splits (even depths) compare x-coordinates
- Horizontal splits (odd depths) compare y-coordinates
- All original points end up as leaves

5.9 Exercise 5.2: BUILDKDTREE Complexity

Problem: (It is enough to give an intuitive idea) Prove that the BUILDKDTREE for a set of n points has running time $O(n \log n)$ and uses $O(n)$ storage.

Solution: Let's break this down into two parts: storage complexity and runtime complexity.

1. Storage Complexity $O(n)$:

- Each splitting line divides points into two equal parts (up to unity)
- For $n = 2^k$ points, we need $2^k - 1$ lines (internal nodes)
- For splitting 2, the total nodes (parents + leaves) is:
$$2^k + 2^{k-1} = n + n/2 = 3n/2 < 3n$$
- For n not a power of 2, there exists t where $2^{t-1} < n < 2^t$
- Number of internal nodes n_p is bounded by:
$$2^{t-2} < n_p < 2^{t-1}$$
- Total nodes (parents + leaves) satisfies:
$$3 \cdot 2^{t-2} < n + n_p < 3 \cdot 2^{t-1}$$
- Therefore: $n + n_p < 3 \cdot 2^{t-1} < 3n$
- Each node uses $O(1)$ storage
- Total storage: $O(1) \cdot O(n) = O(n)$

2. Runtime Complexity $O(n \log n)$:

- BUILDKDTREE is recursive:
  - Each recursion splits n points into two $n/2$ subsets
  - Split cost is linear ($O(n)$): finding median in x or y coordinates
- Building time $T(n)$ follows the recurrence:
$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n > 1 \end{cases}$$
- By Master Theorem (as in Merge-Sort):
  - $a = 2$ (subproblems)
  - $b = 2$ (size reduction)
  - $f(n) = O(n)$ (split cost)
  - Case 1: $f(n) = O(n) = \Theta(n^{\log_2 2}) = \Theta(n)$
  - Therefore: $T(n) = O(n \log n)$

Key Insights:

- Storage is linear because each point becomes exactly one leaf node
- Runtime is $O(n \log n)$ due to:
  - Linear-time splitting at each level
  - Logarithmic number of levels in the tree
- Example of a point p_1
- Example of a point p_2
- Example of a point p_3
- Example of a point p_4
- Example of a point p_5

5.10 Exercise 6.1: Graph Transpose

Problem: The transpose of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(u, v) \mid (v, u) \in E\}$. Describe efficient algorithms for computing G^T from G, for both the adjacency-list and adjacency-matrix representations of G. Analyze the running times of your algorithms.

Solution: Let's analyze both representations:

1. Adjacency Matrix Representation:

- Let $|V| = n$. If M_G is the adjacency matrix of G, then M_G^T is the adjacency matrix of G^T
- For a matrix M:
$$M = \begin{pmatrix} m_{11} & m_{12} & \cdots & m_{1n} \\ m_{21} & m_{22} & \cdots & m_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{n2} & \cdots & m_{nn} \end{pmatrix}$$
- The transpose matrix is:
$$M^T = \begin{pmatrix} m_{11} & m_{21} & \cdots & m_{n1} \\ m_{12} & m_{22} & \cdots & m_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ m_{1n} & m_{2n} & \cdots & m_{nn} \end{pmatrix}$$
- Example:
$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad M^T = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$
- Algorithm:
  - Swap all pairs (m_{ij}, m_{ji}) outside main diagonal
  - Number of swaps: $n^2 - n$ (excluding diagonal)
  - Time complexity: $\Theta(n^2)$

2. Adjacency List Representation:

- Algorithm:
  1. Create new "vertical list" with all vertices V of G^T (same as G)
     - Cost: $\Theta(n)$
  2. For each vertex v in G's adjacency list:
     - For each vertex w in v's list:
       * Add v to w's list in G^T
  3. Total cost: $\Theta(|V| + |E|)$

Complexity Analysis:

- Adjacency Matrix: $\Theta(n^2)$
  - Must process every cell in matrix
  - Independent of number of edges

- Adjacency List: $\Theta(|V| + |E|)$
  - Must process each vertex once
  - Must process each edge once
  - More efficient for sparse graphs

Key Insights:

- For dense graphs ($|E| \approx |V|^2$), both representations have similar performance
- For sparse graphs ($|E| \ll |V|^2$), adjacency list is more efficient
- Space complexity matches time complexity for both representations

5.11 Exercise 6.2: Dijkstra's Algorithm with Negative Weights

Problem: Give a simple example of a directed graph with some negative-weight edges for which Dijkstra's algorithm produces incorrect answers.

Solution: Any directed graph containing a cycle with a negative total weight will cause Dijkstra's algorithm to produce incorrect answers. Here's a simple example:

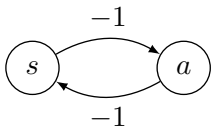


Figure 15: *
A graph with negative-weight cycle

Why This Breaks Dijkstra's Algorithm:

- Dijkstra's algorithm assumes that adding an edge to a path cannot decrease its total weight
- In this example:
  - Initial distance to a: $d[a] = -1$
  - After one cycle: $d[a] = -3$
  - After two cycles: $d[a] = -5$
  - And so on...
- The shortest path is undefined as we can keep traversing the cycle to get arbitrarily small path weights

Key Insights:

- Dijkstra's algorithm is guaranteed to work only with non-negative edge weights
- For graphs with negative weights, use the Bellman-Ford algorithm instead
- A negative cycle is detectable if after $|V| - 1$ iterations, we can still relax an edge
- In practice, negative weights often indicate a modeling error in the problem

5.12 Exercise 6.3: Dijkstra's Algorithm Iterations

Problem: We apply DIJKSTRA (Lecture 13 and Lecture 14) to the following graph (G, V) with starting node a. After INITIAL-SINGLE-SOURCE(G, a) we have $a.d = 0$ and $v.d = \infty$ for each vertex $v \neq a$. What are the distances $c.d$, $f.d$, and $d.d$ after two iterations?

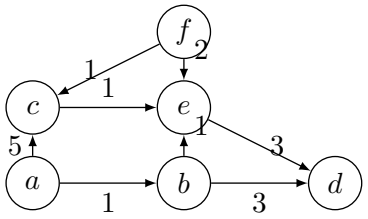


Figure 16: *
Graph for Dijkstra's Algorithm

Solution: Let's trace through the first two iterations:

Initial State:

- $a.d = 0$ (source)
- All other vertices: $v.d = \infty$
- $Q = \{a, b, c, d, e, f\}$

First Iteration:

- Extract-Min: vertex a ($d = 0$)
- Relax edges from a:
  - (a, b): $b.d = \min(\infty, 0 + 1) = 1$
  - (a, c): $c.d = \min(\infty, 0 + 5) = 5$
- $Q = \{b, c, d, e, f\}$

Second Iteration:

- Extract-Min: vertex b ($d = 1$)
- Relax edges from b:
  - (b, d): $d.d = \min(\infty, 1 + 3) = 4$
  - (b, e): $e.d = \min(\infty, 1 + 1) = 2$
- $Q = \{c, d, e, f\}$

Final Answer:

- $c.d = 5$ (from first iteration)
- $f.d = \infty$ (not yet reached)
- $d.d = 4$ (from second iteration)

Key Insights:

- Dijkstra's algorithm processes vertices in order of increasing distance
- After each iteration, distances are final for the processed vertex
- Unreachable vertices maintain distance ∞
- Each edge relaxation potentially updates the tentative distance to a vertex

5.13 Exercise 7.1: Sorting by Polar Angle

Problem: The polar angle of a point p_i with respect to an origin p_0 is the angle from the semi-horizontal straight line r and the vector $\overrightarrow{p_0p_i}$. The positive direction of the angle is counterclockwise. Angles are in the interval $[0, 2\pi)$. Write pseudocode to order n points $q_1, \ldots, q_n$ by their polar angles in increasing order, with $O(n \log n)$ running time.

Solution Strategy:

1. Let $p_0 = [x_0, y_0]$ and $q_i = [x_i, y_i]$.
2. Divide points $A = \{q_1, \ldots, q_n\}$ into:
   - A_1: Points with $y \geq y_0$
   - A_2: Points with $y < y_0$
3. Polar angles in A_1 are in $[0, \pi]$, in A_2 are in $(\pi, 2\pi)$.
4. Use ANGLEMEREGE-SORT to sort both A_1 and A_2.

Algorithm Details:

- PARTITIONANGLE:
  - Purpose: Divides the array into two groups based on the y-coordinate relative to y_0.
  - Process: Iterates through the array, swapping elements to ensure points in A_1 have polar angles in $[0, \pi]$ and points in A_2 in $(\pi, 2\pi)$.
  - Complexity: $O(n)$, as each point is processed once.
- ANGLEMERGE-SORT:
  - Purpose: Sorts points within each subset based on polar angles.
  - Process: Uses a modified merge sort where the cross product is used to compare angles, ensuring correct order.
  - Complexity: $O(n \log n)$, typical for merge sort algorithms.
- ANGLE-SORT:
  - Purpose: Integrates the above methods to sort the entire set A.
  - Process: Calls PARTITIONANGLE to separate the points, then applies ANGLEMEREGE-SORT to each subset.
  - Complexity: $O(n \log n)$, combining the efficiencies of partitioning and sorting.

Pseudocode:

```
procedure PARTITIONANGLE(A, y0):
    i = 0
    for j = 1 to n do
        if y.A[j] >= y0 then
            i = i + 1
            exchange A[i] with A[j]
    return i + 1

procedure ANGLEMEREGE-SORT(A, p, r):
    if p < r then
        q = (p + r) / 2
        ANGLEMEREGE-SORT(A, p, q)
        ANGLEMEREGE-SORT(A, q + 1, r)
        ANGLEMEREGE(A, p, q, r)

procedure ANGLE-SORT(A):
    if A != NIL then
        n = length.A
        q = PARTITIONANGLE(A, x0)
        ANGLEMEREGE-SORT(A, 1, q - 1)
        ANGLEMEREGE-SORT(A, q, n)
```

Complexity Analysis:

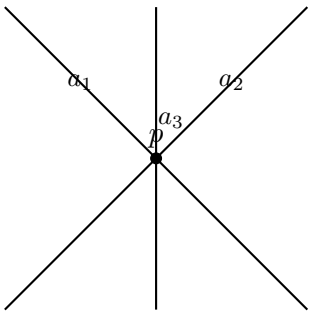
- PARTITIONANGLE: $O(n)$
- ANGLEMEREGE-SORT: $O(n \log n)$
- Overall complexity: $O(n \log n)$

Key Insights:

- Sorting by polar angle requires careful handling of angle wrap-around.
- Efficient sorting is crucial for applications in computational geometry.
- Polar angles can be computed using atan2$(y_i - y_0, x_i - x_0)$.

5.14 Exercise 7.3: ANY-SEGMENTS-INTERSECT

Problem: Argue that `ANY-SEGMENTS-INTERSECT` works correctly even if three or more segments intersect at the same point.



Solution Strategy:

1. Identify the conditions under which `ANY-SEGMENTS-INTERSECT` returns true.
2. Analyze the behavior of the algorithm when three or more segments intersect.
3. Prove correctness by considering event points and the sweep line.

Algorithm Details:

- **Intersection Detection:**
  - Inserts a segment when it intersects with the above or below segment.
  - Deletes a segment when its adjacent segments meet at a point.
- **Event Point Analysis:**

- Considers the first intersection point p of n segments.
- Analyzes the left and right points of segments at p.

Pseudocode Explanation: The pseudocode iterates through each event point, which represents either the start or end of a segment. When a left endpoint is encountered, the segment is added to the active set. If a right endpoint is reached, the segment is removed. The algorithm checks for intersections by evaluating if three or more segments meet at a point, returning true if so.

Complexity Analysis:

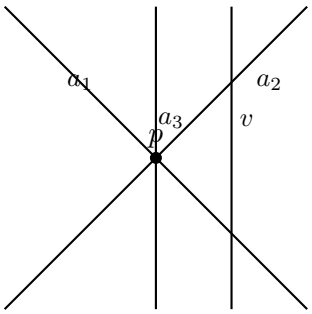
- Event processing: $O(n \log n)$
- Intersection checks: Efficient for multiple segments

Key Insights:

- The algorithm correctly identifies intersections involving multiple segments.
- The sweep line approach efficiently handles complex intersections.

5.15 Exercise 7.4: ANY-SEGMENTS-INTERSECT with Vertical Segments

Problem: Show that `ANY-SEGMENTS-INTERSECT` works correctly in the presence of vertical segments if we treat the bottom endpoint of a vertical segment as if it were a left endpoint and the top endpoint as if it were a right endpoint. How does your answer to Exercise 3 change if we allow vertical segments?



Solution Strategy:

1. Treat vertical segments with special rules for endpoints.
2. Analyze how vertical segments affect the intersection detection.
3. Prove correctness by considering vertical and non-vertical interactions.

Algorithm Details:

- **Vertical Segment Handling:**
  - Treat the bottom endpoint as a left endpoint.
  - Treat the top endpoint as a right endpoint.
- **Intersection Detection:**
  - Check intersections with vertical segments using the sweep line.
  - Ensure correct ordering of event points.

Pseudocode Explanation: The pseudocode processes each event point, treating vertical segment endpoints according to their roles. By treating the bottom as a left endpoint and the top as a right endpoint, the algorithm maintains the correct order and checks for intersections. This adjustment ensures that vertical segments are handled without altering the fundamental logic of the sweep line approach.

Impact of Vertical Segments: Allowing vertical segments in Exercise 7.3 would require treating their endpoints as described, ensuring that intersections involving vertical segments are detected correctly. The algorithm's logic remains intact, with the addition of handling vertical segment endpoints appropriately.

Complexity Analysis:

- Event processing: $O(n \log n)$
- Intersection checks: Efficient for vertical and non-vertical segments

Key Insights:

- The algorithm handles vertical segments by adjusting endpoint roles.
- The sweep line approach remains effective for complex intersections.

6 Exercises Part 2

6.1 Exercise 8.1: Drawing 5 Lines

Problem: Prove that you cannot draw 5 lines on the Euclidean plane in such a way that each line cuts exactly 3 other lines.

Solution: Using graph modeling:

- One line = one node
- One intersection = one edge
- Problem: Draw a graph with 5 nodes, each of degree 3
- Sum of the degrees = 15
- In any graph, sum of degrees must be even (each edge has 2 extremities)
- Conclusion: Infeasible!

Explanation: To solve similar problems, consider the following steps:

1. Model the problem using graph theory concepts.
2. Identify nodes and edges based on the problem statement.
3. Calculate the degree of each node and ensure the sum of degrees is even.
4. Use these calculations to determine feasibility.

6.2 Exercise 8.2: TSP Mathematical Formulation

Problem: Give an exact formulation of the travelling salesperson problem.

Solution: Input data:

- Distance matrix $D = (d_{ij})$ between cities i and j
- Objective: Find permutation p of n cities minimizing:

$$\sum_{i=1}^{n-1} d_{p_i p_{i+1}} + d_{p_n p_1}$$

Explanation: For similar exercises:

1. Understand the problem constraints and objectives.
2. Define the input data clearly, such as distance matrices.
3. Formulate the objective function based on the problem requirements.
4. Use permutations or combinations to find optimal solutions.

6.3 Exercise 8.3: Cards Problem Optimization

Problem: 50 cards numbered 1 to 50 must be separated into 2 stacks. First stack sum = 1170, second stack product = 36000.

Solution: Encoding:

- Zero-one vector s where $s_i = 0$ means card i is in first stack
- Objective function to minimize:

$$f(s) = \left| \frac{1170 - \sum_{i=1}^{50} i \cdot (1 - s_i)}{1170} \right| + \left| \frac{36000 - \prod_{i=1}^{50} i^{s_i}}{36000} \right|$$

Explanation: For similar optimization problems:

1. Define the decision variables and constraints clearly.
2. Formulate the objective function to reflect the problem goals.
3. Use mathematical tools to solve for optimal solutions.
4. Validate results by checking against constraints.

6.4 Exercise 8.4: Timetable for Exams

Problem: Timetable for Exams

Solution: This problem can be formulated as a Vertex Colouring Problem: Each examination is a node, and two nodes are connected if at least one student exists who has to take both exams.

Steps to Solve:

1. **Identify Nodes:** Each exam is a node in the graph.
2. **Connect Nodes:** Draw an edge between nodes if a student is enrolled in both exams.
3. **Apply Colouring:** Use colouring to assign days to exams such that no two connected nodes share the same colour.
4. **Minimize Colours:** The goal is to use the minimum number of colours, representing the minimum number of days.

6.5 Exercise 8.5: Permutation Flow Shop Problem

Problem: Number of Solutions for Permutation Flow Shop Problem

Solution: For Permutation Flow Shop Problems, only the ordering of the n jobs on the first machine need to be selected, since the ordering of tasks on the other machines is then fixed. So there are n! permutations of n jobs.

Steps to Solve:

1. **Understand the Problem:** Recognize that the order of jobs on the first machine determines the sequence for all machines.
2. **Calculate Permutations:** Use factorial $n!$ to calculate the number of possible job sequences.
3. **Apply Constraints:** Consider any additional constraints that might affect the ordering.

6.6 Exercise 8.6: Asymptotic Runtime

Problem: Asymptotic Runtime

Solution: a) The asymptotic runtime of the given code is $O(n^2)$ b) The ascending ordering is: 1, log n, n, n^2, n^3, $(3/2)^n$, 2^n.

Steps to Solve:

1. **Analyze Code:** Break down the loops to understand their contribution to runtime.
2. **Identify Dominant Terms:** Focus on the terms that grow fastest as n increases.
3. **Order Functions:** Arrange functions by growth rate to understand their asymptotic behavior.

6.7 Exercise 8.7: Dijkstra's Algorithm

Problem: Example on Dijkstra's Algorithm

Solution: A, B, C, D, E lambda = (0, 5, 2, 10, 12) p = (-, C, A, B, D)

Steps to Solve:

1. **Initialize Distances:** Set the starting node's distance to zero and all others to infinity.
2. **Select Node:** Choose the node with the smallest tentative distance.
3. **Update Neighbors:** Calculate the tentative distances for neighboring nodes.
4. **Repeat:** Continue until all nodes are processed.
5. **Construct Path:** Use the predecessor array to reconstruct the shortest path.

6.8 Exercise 8.8: Prim's Algorithm

Problem: Example on Prim's Algorithm

Solution: A, B, C, D, E, F, G lambda = (0, 7, 5, 5, 7, 6, 9) p = (-, A, E, A, B, D, E) E_T = {(A,D), (D,F), (A,B), (B,E), (E,C), (E,G)}

Steps to Solve:

1. **Initialize:** Start with a single node and an empty edge set.
2. **Select Edge:** Choose the smallest edge connecting the tree to a new vertex.
3. **Add Edge:** Include this edge and vertex in the tree.
4. **Repeat:** Continue until all vertices are included in the tree.

6.9 Exercise 9.1: Time Complexity of TSP Heuristics

Problem:

a) What is the complexity of the Nearest Neighbor heuristic for the TSP?

b) Same question for the pilot method, if the Nearest Neighbor heuristic is used as pilot.

c) Same question for a beam search procedure, if B branches are retained at each level, and the tree is examined up to k levels ahead, but the retained branches are only extended by one node per step.

Solution:

a) Nearest Neighbor has time complexity $O(n^2)$, since there are $n-1$ cities that need to be added to the tour, and at each step the nearest neighbor (to the last city added) out of the j remaining cities has to be computed, which takes $O(j)$ (with $j = n-1, n-2, \ldots, 2, 1$) for each city. So it's $O((n-1)+(n-2)+\ldots+2+1) = O(n^2)$.

b) The Pilot Method with Nearest Neighbor has time complexity $O(n^4)$, since there are $n-1$ cities that need to be added to the tour, and at each step, the best partial tour so far has to be extended in j different ways by one of the j remaining cities each, and then using the Nearest Neighbor strategy. According to Subtask a) (with $j = n-1, n-2, \ldots, 2$) we get $O((n-1)*(n-2)^2)+O((n-2)*(n-3)^2)+\ldots+O(2*1^2) = O(n^4)$.

c) Beam Search has time complexity $O(B * n^{k+1})$, since there are $n-1$ cities that need to be added to the tour. In each step, the B best partial tours so far are extended by k cities each, out of all the j remaining cities (with $j = n-2, n-3, \ldots, k$). So it's $O(B*(n-2)^k)+O(B*(n-3)^k)+\ldots+O(B*k^k) = O(B*n^{k+1})$. Note, that this is an upper bound which is the more pessimistic, the larger k is chosen in comparison to n.

6.10 Exercise 9.2: Comparing Heuristics for TSP

Problem:

a) Implement the Nearest Neighbor heuristic for TSP. Note the following:

- A stub `tsp_nearest_neighbor.py` is provided in the Python framework in the subfolder `Python -> heuristics`. This stub is integrated in the runnable main demo script `demo_tsp_nearest_neighbor.py` on the top-level folder `Python`.
- There are several ready-to-use TSP instances provided in the subfolder `Python -> heuristics -> problems -> tsp -> instances`, which may be selected in the main demo script `demo_tsp_nearest_neighbor.py`.
- Check the file `instance.py` in subfolder `Python -> heuristics -> problems -> tsp -> utils`, to get familiar with the data structure used to represent a TSP instance.

b) Compare the computational time and the quality of the solutions of:

- Nearest Neighbor (from part a),
- Random Sampling (provided ready-to-use: `demo_tsp_random_sampling.py`),
- Random Best Insertion (provided ready-to-use: `demo_tsp_greedy_insertion_random.py`).

6.11 Exercise 9.3: "Good" Algorithm for TSP

Problem: Implement a good heuristic algorithm for TSP. You can develop and implement your own ideas, or use some of the methods presented in the lecture so far.

Evaluate your heuristic on the provided instances data sets in the Python framework in subfolder `Python -> heuristics -> problems -> tsp -> instances`.

6.12 Exercise 10.1: Manual Local Searches

Problem: An integer function $[-7, 7] \times [-6, 7] \to [-10, 754]$ is explicitly given in a table. The moves consist of modifying a single variable by $+/-$ one unit. The moves are ordered as follows: $(+1, 0), (0, +1), (-1, 0)$ and $(0, -1)$. The goal is to minimize the function value by means of a local search using:

a) First Improving Move strategy, starting from the upper right corner $(x = 7, y = -6)$.

b) Best Improving Move strategy, starting from upper left corner $(x = -7, y = -6)$.

Steps to Solve:

1. **Understand the Move Strategies:**
   - **First Improving Move:** Choose the first move that improves the function value.
   - **Best Improving Move:** Evaluate all possible moves and choose the one that provides the greatest improvement.
2. **Apply the Strategies:**
   - Start at the specified corner and apply the move strategy.
   - For First Improving Move, stop as soon as an improvement is found.
   - For Best Improving Move, evaluate all moves and select the best one.
3. **Iterate:** Repeat the process until no further improvements can be made.
4. **Analyze Results:** Compare the paths and final values obtained by each strategy.

6.13 Exercise 10.2: 2-opt and 3-opt Moves for TSP

Problem: For a Travelling Salesperson Problem instance with 8 cities, consider the tour:
`c1-c2-c3-c4-c5-c6-c7-c8-c1.`

a) Give the resulting tour of applying the 3-opt move, where the edges c2-c3 and c4-c5 and c6-c7 are replaced, such that no sub trail is reversed.

b) Same as in a), but this time the two shortest sub trails are reversed.

c) Give the resulting tour of applying the 2-opt move, where the edges c2-c3 and c6-c7 are replaced, such that the sub trail containing c1 is not reversed.

Solution:

a) `c1-c2-c5-c6-c3-c4-c7-c8-c1.`

b) `c1-c2-c4-c3-c6-c5-c7-c8-c1.`

c) `c1-c2-c6-c5-c4-c3-c7-c8-c1.`

6.14 Exercise 10.3: Greedy for Knapsack Problem

Problem: The Knapsack Problem is defined as follows: Given a set of n items numbered from 1 up to n, each with a weight w_i and a value v_i, along with a maximum weight capacity W of the knapsack,

$$\max \sum_{i=1}^{n} v_i x_i \quad \text{subject to} \quad \sum_{i=1}^{n} w_i x_i \leq W \quad \text{and} \quad x_i \in \{0, 1\}$$

a) Consider a "Naïve Greedy" algorithm, which always takes the item with maximum value that still fits into the knapsack. Show that this algorithm can create arbitrarily bad solutions.

b) Algorithm "Smarter Greedy" works as follows: compute v_i/w_i for each item, and sort all items in descending order. Then take items in this order as long as they fit into the knapsack. This algorithm can create good solutions, but also bad ones. Show that there exists an example with two items where the solution is arbitrarily bad.

c) What are reasonable neighbourhoods for the Knapsack Problem?

Solution:

a) Let $n = W$ and $v_1 = 2, w_1 = W, v_2 = v_3 \ldots = v_n = 1, w_2 = w_3 = \ldots = w_n = 1$. Then Naive Greedy takes item 1 with value 2, whereas a better solution would be to take all other items with value $n - 1$.

b) Worst case for Smarter Greedy: Let $v_1 = 2, w_1 = 1, v_2 = W$ and $w_2 = W$. Then Smarter Greedy takes item 1, but item 2 would be much better.

c) Potential Neighbourhoods:
   - Flip one bit x_i, if result is a valid solution.
   - Exchange one item with another, if result is a valid solution.

6.15 Exercise 10.4: Knapsack Problem with Simulated Annealing

Problem: For the Knapsack Problem instance Knapsack_P08, find the optimal solution using Simulated Annealing. The instance is defined as follows:

- $N = 24$
- $W = 6404180$
- Weights: 382745, 799601, 909247, 729069, 467902, 44328, 34610, 698150, 823460, 903959, 853665, 551830, 610856, 670702, 488960, 951111, 323046, 446298, 931161, 31385, 496951, 264724, 224916, 169684
- Values: 825594, 1677009, 1676628, 1523970, 943972, 97426, 69666, 1296457, 1679693, 1902996, 1844992, 1049289, 1252836, 1319836, 953277, 2067538, 675367, 853655, 1826027, 65731, 901489, 577243, 466257, 369261

Solution: Optimal Solution for Instance Knapsack_P08:

- Value = 13549094
- Items: 1 1 0 1 1 1 0 0 1 1 0 1 0 0 1 0 0 0 0 1 1 1

6.16 Exercise 11.1: Manual Tabu Search

Problem: An integer function $[-7, 7] \times [-6, 7] \to [-10, 754]$ is explicitly given in a table. A Tabu Search is used to find the minimum of this function. The neighbourhood consists of increasing or decreasing the value of one variable by 1. The tabu condition forbids changing back a variable that has been changed at most t iterations before. In every iteration, the best possible move is taken, even if it increases the function value. The search stops if no allowed move exists or if 25 iterations have been performed.

a) Start a tabu search with $t = 3$ at $(x, y) = (-7, 7)$.

b) Start a tabu search with $t = 1$ at $(x, y) = (-7, -6)$.

Steps to Solve:

1. **Understand the Tabu Condition:**
   - Track moves and prevent reversing changes made within the last t iterations.
2. **Apply the Search:**
   - Start at the specified point and apply the best move strategy.
   - Record each move and apply the tabu condition.
3. **Iterate:** Continue until no further moves are allowed or 25 iterations are reached.
4. **Analyze Results:** Compare the paths and final values obtained by each strategy.

6.17 Exercise 11.2: Tabu Search on Knapsack Problem

Problem: Maximize $11x_1+10x_2+9x_3+12x_4+10x_5+6x_6+7x_7+5x_8+3x_9+8x_{10}$ subject to $33x_1+27x_2+16x_3+14x_4+29x_5+30x_6+31x_7+33x_8+14x_9+18x_{10} \leq 100$

$x_i \in \{0, 1\}, i = 1, \ldots, 10$

Initial solution: $(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$, revenue: 0

Move: flip a bit $x_i \leftarrow 1 - x_i$

Tabu condition: don't flip the same bit for 3 (complete) iterations

Tabu list: iteration AFTER which a bit can be flipped again

Initial tabu list: $(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

Solution: Compute the 10 first iterations using the greedy strategy:

- Insert the object with maximum value that fits (breaking ties by choosing the object weighing less).
- If no further object can be inserted, remove the object with minimal value (breaking ties by choosing the heavier object).

6.18 Exercise 11.3: Neighbourhood Analysis for CVRP

Problem: Analyse the properties:

- Size of Neighbourhood
- Connectivity
- Run time complexity for one move

for the following neighbourhoods for the Capacitated Vehicle Routing Problem (CVRP):

a) Neighborhood description: Select customer X randomly – Select destination tour randomly – Select insertion position for X in destination tour randomly.

b) Neighborhood description: Select customer X randomly – Select destination tour randomly – Select best insertion position for X in destination tour.

Solution:

a) Properties:
   - Size of Neighborhood: $O(n \cdot (\sum_{i=1}^{m}(1 + \text{number of cust. in tour } T_i))) = O(n^2)$.
   - Connectivity: No, since the number of tours cannot change.

- Run Time for one move: $O(1)$, since only randomly chosen vertices and the corresponding edges in the original and destination tour are changed.

b) Properties:

- Size of Neighborhood: $O(n \cdot m) = O(n^2)$ (worst case: every customer is in a tour on his own).
- Connectivity: No, since the number of tours cannot change.
- Run Time for one move: $O(\text{number of cust. in destination tour}) = O(n)$, since best insertion position has to be found in destination tour (worst case: all customers in a single tour).

6.19 Exercise 12.1: Ant Colony Optimization for TSP

Problem: If a meta heuristic for a certain problem shall be implemented, then typically there are several implementations already available on the internet. In this exercise, your task is to find, evaluate and optimize one of these.

a) Search an existing implementation of ACO for TSP from the internet.

b) Evaluate the performance and stability of the tool for various instances (e.g. by comparing solutions to those computed with the algorithms from previous exercises).

c) Experiment with various settings for the parameters of your ant system.

d) Apply your "best" setting for the selected algorithm to the standard instances of TSP (see Exercises of Lecture 2) and report your results at:

Solution: An implementation of ACO for TSP can be found e.g. at:
`https://github.com/ppoffice/ant-colony-tsp`

6.20 Exercise 13.1: Order-1 and Partially Mapped Crossover

Problem: Apply the following crossover techniques to two permutations, swapping positions 4-7 (in blue):

a) Apply Order-1 Crossover to the following two permutations:

P1: 1 4 2 8 5 7 3 6 9

P2: 7 5 3 1 9 8 6 4 2

b) Apply Partially Mapped Crossover to the following two permutations, swapping positions 4-7 (in blue).

P1: 1 4 2 8 5 7 3 6 9

P2: 7 5 3 1 9 8 6 4 2

Solution:

a) Order-1 Crossover

- **O1:** 5 7 3 1 9 8 6 4 2
- **O2:** 1 9 6 8 5 7 3 4 2

b) Partially Mapped Crossover

- **O1:** 7 4 2 1 9 8 6 3 5
- **O2:** 1 9 6 8 5 7 3 4 2

6.21 Exercise 13.2: Genetic Algorithm for TSP

Problem: If you want to implement a meta heuristic for a certain problem, there are typically several implementations already available on the internet. In this exercise, your task is to evaluate and optimize one of these.

a) Search an existing implementation of a Genetic Algorithm for TSP from the internet.

b) Evaluate the performance and stability of the tool for various sample data (using e.g. the random generator for new solutions from the previous exercises).

c) Experiment with various settings for the parameters of your system.

d) Apply your "best" setting for the selected algorithm to the standard instances of TSP (see Exercises of Lecture 2) and report your results at:

`https://docs.google.com/forms/d/e/1FAIpQLSdUJ`
`viewform?usp=sf_link`

Solution: An implementation of GA for TSP can be found e.g. at:
`https://github.com/maoai/tsp-genetic-python`

7 Mock Exam Part 1

7.1 Exercise 1: Running Time

Example 1

1. (6P) Write the following Theta-classes in non-decreasing order from left to right (i.e., smallest on

$\Theta(n^2), \Theta(e^{\log_2 n}), \Theta(\sin n), \Theta(\ln^2 n), \Theta(\log_2 n), \Theta(n^3 + \ln(n^2)), \Theta(e^{n^2}), \Theta(e^{\ln(n^4)})$

Solution: Understand growth rates: $\sin n$ is slowest, followed by logarithmic, polynomial, and exponential. Use limits to compare and order.

2. (3P) We consider a recursive algorithm that involves an input with n objects and having running time $T(n) = 4 \cdot T(n/2) + n^2$. Determine the running Theta class of the algorithm.

Solution: Identify recurrence: $T(n) = a \cdot T(n/b) + f(n)$. Apply Master Theorem: Compare $f(n)$ with $n^{\log_b a}$ and determine the case.

3. (3P) We consider a recursive algorithm that involves an input with n objects and having running time $T(n) = 4 \cdot T(n/2) + n$. Determine the running Theta class of the algorithm.

Solution: Apply Master Theorem: Use Case 1 for $f(n) = O(n^{\log_b a - \epsilon})$.

4. (3P) We consider a recursive algorithm that involves an input with n objects and having running time $T(n) = 4 \cdot T(n/2) + n^3$. Determine the running Theta class of the algorithm.

Solution: Apply Master Theorem: Use Case 3 for $f(n) = \Omega(n^{\log_b a + \epsilon})$.

Solution:

1. $\Theta(\sin n), \Theta(\log_2 n), \Theta(\ln^2 n), \Theta(n^2), \Theta(e^{\log_2 n}), \Theta(n^3 + \ln(n^2)), \Theta(e^{\ln(n^4)}), \Theta(e^{n^2})$

2. Case 2 Master theorem. $\Theta(n^2 \ln n)$.

3. Case 1 Master theorem. $\Theta(n^2)$.

4. Case 3 Master theorem. $\Theta(n^3)$. (In all three cases, 1p corrected answer 2p correct justification)

Example 2

1. (4P) Write in increasing order from left to right the following list of Theta classes. When two classes are equal, make it clear.

$\Theta(n^{4/5}), \Theta(100n), \Theta(n \log_4 n), \Theta(n^{3/2}), \Theta(n^{\sqrt{n}}), \Theta(e^{\sqrt{2} \cdot n}), \Theta(e^n), \Theta(n!)$

Solution: Understand growth rates: Familiarize yourself with common functions like polynomials, logarithms, exponentials, and factorials. Use limits to compare and order.

2. (2P) Give two functions $f(n)$ and $g(n)$ such that $\Theta(f(n) + g(n)) \neq \Theta(f(n))$ and $\Theta(f(n) + g(n)) \neq \Theta(g(n))$

Solution: Choose functions with different growth rates.

Solution:

$\Theta(n^{4/5}), \Theta(100n), \Theta(n \log_4 n), \Theta(n^{\sqrt{n}}), \Theta(n^{3/2}) = \Theta(n^{\sqrt{n}}), \Theta(e^n), \Theta(e^{\sqrt{2} \cdot n}), \Theta(n!)$

$f(n) = n$ and $g(n) = -n + 1$

Exercise 2: Pseudocode Examples

Example 1: Pseudocode

1. (6P) Write a pseudocode, which takes the input $A = \{a_1, a_2, \ldots, a_n\}$ and gives you as output $B = \{a_n, \ldots, a_2, a_1\}$ (i.e. same entries as in A but, in the reversed order).

Solution approach: - Use array indexing to swap elements - Only need to iterate through half the array (why? because each swap handles two positions) - Track running time: each operation is constant time, done n/2 times

```
REVERSEARRAY(A)
for i = 1 to n/2
    exchange A[i] with A[n-i+1]
return A
```

Running time: $\Theta(n)$ because we perform n/2 constant-time operations.

2. (9P) Let S be a set with n integers, randomly ordered and not necessarily distinct. Let m be an integer. Write a pseudocode, which tests whether there are two elements $a, b \in S$ with $a + b = m$ (a and b may be the same number).

Solution approach: - Naive solution: Check all pairs with nested loops = $O(n^2)$ - Better solution: Sort first, then use two pointers = $O(n \log n)$ - Best solution: Single pass with smart pointer movement = $O(n)$

Naive solution (for understanding):

```
SUMTEST(S,m)
for i = 1 to n-1
    for j = i+1 to n
        if S[i] + S[j] = m return true
return false
```

Optimized solution:

```
SUMTESTBEST(S,m)
i = 1 to j = n
if S[i] + S[j] = m return true
if S[i] + S[j] < m, i=i+1
else j = j+1 (# that is if S[i] + S[j] > m)
stop when returns true or i > j
```

Key insights for optimization: - Avoid checking all pairs - Use array properties to skip unnecessary checks - Think about how to move pointers based on sum comparison with target - Running time improves from $O(n^2)$ to $O(n)$

Exercise 3: Sweep Line Algorithm

Example 1: ANYSEGMENTINTERSECT

Given the algorithm ANYSEGMENTINTERSECT and line segments on an (x,y)-plane, determine the sweep lines and partial orders.

Solution approach: 1. Understand the algorithm: - Sorts endpoints from left to right - Maintains partial order of segments crossing sweep line - Checks for intersections at each event point

2. Steps to solve: a) First, identify all endpoints (left and right) of segments b) Sort them from left to right c) For each vertical sweep line at these points: - Draw the vertical line - List segments crossing this line from bottom to top - Check for intersections between adjacent segments

3. Key points to remember: - Left endpoint: INSERT segment into order - Right endpoint: DELETE segment from order - Check ABOVE and BELOW neighbors for intersections - Stop when intersection found

4. Example solution format: Sweep line at x = -2: Segments (bottom to top): a, f

Sweep line at x = -1: Segments: a, e, f, b

Sweep line at x = 0: Segments: c, e, f, b

Running time analysis: - Sorting endpoints: $O(n \log n)$ - Each endpoint processed once: $O(n)$ - Total: $O(n \log n)$

Common mistakes to avoid: - Don't forget to order segments from bottom to top - Check both ABOVE and BELOW at insertion - List ALL segments crossing each sweep line

Exercise 4: Points on the Plane

Example 1: Points on the Plane

1. (9P) Check for collinear points in 2D plane:

Solution steps: a) For each point p_0 as center: - Calculate angles between $\overline{p_0p_1}$ and all other segments $\overline{p_0p_j}$ - Use DIRECTION(p_0, p_i, p_j) = cross product $(p_1 - p_0) \times (p_j - p_0)$ - Calculate angle: $\alpha_j = \sin^{-1}(\frac{\text{cross product}}{||p_1-p_0|| \cdot ||p_j-p_0||})$

b) For each center point: - Sort angles using merge sort: $O(n \log n)$ - Use SUMTESTBEST to find angles with 0 or π difference: $O(n)$ - Three points collinear if difference is 0 or π

Total running time: $O(n^2 \log n)$ because: - Repeat for each point: $O(n)$ - For each center: $O(n \log n)$ for sorting + $O(n)$ for checking - Final complexity: $n \cdot O(n \log n) = O(n^2 \log n)$

2. (6P) BUILDKDTREE construction:

Steps to construct KD-Tree: a) Start with depth 0 (vertical split) - Find median x-coordinate - Split points into left/right sets

b) At depth 1 (horizontal split): - Find median y-coordinate in each subset - Split into top/bottom sets

c) Continue alternating between: - Even depth: vertical splits (x-coordinate) - Odd depth: horizontal splits (y-coordinate)

Example tree structure:

```
At root (depth 0): vertical split
- Left child: points left of median
- Right child: points right of median
  At depth 1: horizontal splits
  - Top/bottom division for each subset
```

Key insights: - Alternate between x and y coordinates based on depth - Always split through median point - Each split divides remaining points roughly in half - Tree will be balanced if splits are perfect medians

Exercise 5: Fibonacci Sequence Analysis

Input: Positive integer $n > 2$

Algorithm analysis:

```
INPUT: a positive integer n > 2
Create A an empty array with n entries
Set A[1] = A[2] = 1
for (i = 3; i  n) do
    A[i] = A[i-1] + A[i-2]
return A[n]
```

Solution steps: - Algorithm builds Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, ... - Each number is sum of previous two - For n = 5, output is (1, 1, 2, 3, 5) - Running time: $\Theta(n)$ as we do n-3 iterations with constant operations

Exercise 6: Binary Search Implementation

Write a recursive algorithm (divide-and-conquer) to check if k is in sorted array A.

Solution approach: 1. Decomposition: - Find middle element - Compare k with middle - Determine which half to search

2. Implementation:

```
BINARYSEARCH(A, k, left, right)
    if left > right return -1
    mid = (left + right)/2
    if A[mid] = k return mid
    if A[mid] > k
        return BINARYSEARCH(A, k, left, mid-1)
    return BINARYSEARCH(A, k, mid+1, right)
```

Running time: $\Theta(\log n)$ because: - Each step divides problem size by 2 - Constant work at each level - Tree height is $\log n$

Exercise 7: Minimum Distance Algorithm

Algorithm analysis for finding minimum distance between points:

```
INPUT: n  2 points (x,y), (x,y) ..., (x,y)
a = ((x-x)² + (y-y)²)
for (i = 1; i < n) do
    for (j = i+1; j  n) do
        r = ((x-x)² + (y-y)²)
        if (r < a) then
            a = r
return a
```

Solution analysis: - Computes distance between all pairs of points - Updates minimum distance when smaller distance found - Running time: $\Theta(n^2)$ due to nested loops - Space complexity: $\Theta(1)$ as only storing minimum distance

Exercise 8: Dijkstra's Algorithm

Example 1: Dijkstra's Algorithm

Consider weighted oriented graph with adjacency matrix:

$$\begin{pmatrix} & s & x & y & z & w \\ s & 0 & 2 & 3 & 0 & 0 \\ x & 0 & 0 & 0 & 1 & 5 \\ y & 0 & 0 & 0 & 4 & 0 \\ z & 0 & 0 & 0 & 0 & 3 \\ w & 0 & 0 & 0 & 0 & 2 \end{pmatrix}$$

Steps for Dijkstra's Algorithm: 1. Initialize: - Set source distances (s: 0, others:) - Q = V (all vertices in queue) - S = (empty set of processed vertices)

2. Main loop: - Extract min from Q - Add to S - Update neighbors' distances - Track predecessors

3. Key points to remember: - Black vertices: in S (processed) - White vertices: in Q (unprocessed) - Grey vertex: current EXTRACT-MIN(Q) - Shaded edges: show predecessor values

Exercise 9: KD-Tree Construction

Example 1: KD-Tree Construction

Given points in xy-plane: $(-5,5),(-4,3),(-3,-5),(-2,-1),(-1,0),(0,-3),(1,2),(2,-4),(3,6)$

Steps to construct KD-Tree: 1. Depth 0 (vertical split): - Sort by x-coordinate - Find median x value - Split points into left/right sets

2. For each subset at depth 1: - Sort by y-coordinate - Split into top/bottom

3. Implementation tips: - Even depth: vertical split (x-coord) - Odd depth: horizontal split (y-coord) - Always include median in split - Balance tree by choosing true median

4. Visualization: - Draw vertical/horizontal lines at splits - Show points in resulting regions - Connect nodes based on splits - Label depth and split direction

Example 7: Array Operations

1. Check for duplicates in array:

```
DUPLICATES(A)
    found = false
    for i from 1 to n-1
        for j=i+1 to n
            if a[i]=a[j] found=true
    return found
```

Running time: $\Theta(n^2)$ because we check $\frac{n(n-1)}{2}$ pairs

2. Recursive sum of array segment:

```
Sum(a,l,r)
    if l=r
        return a[l]
    else
        centre=(l+r)/2
        sum1 = Sum(a,l,c)
        sum2 = Sum(a,c+1,r)
        return sum1+sum2
```

Running time analysis: - T(1) = c (constant time) - T(n) = 2T(n/2) + O(1) - By Master Theorem (Case 1): T(n) = (n)

Example 8: Binary Search Tree Operations

Tree-Delete operation steps: 1. Find node to delete 2. If leaf: simply remove 3. If one child: replace with child 4. If two children: - Find successor (minimum in right subtree) - Replace node with successor - Delete successor from original position

Key insights: - Maintain BST properties after deletion - Handle all cases (0, 1, 2 children) - Track parent pointers for clean deletion

Example 9: Segment Intersections

For segments between two parallel lines (y=0 and y=1):

Algorithm approach: 1. Sort points by x-coordinate 2. Count inversions to find intersections 3. Use divide-and-conquer: - Split into halves - Count inversions in each half - Merge and count cross-inversions

Running time analysis: - T(n) = 2T(n/2) + O(n) - By Master Theorem: T(n) = O(n log n)

Key optimization: - Sort once at beginning - Use merge-sort principle for counting - Avoid checking all pairs (would be $O(n^2)$)

8 Mock Exam Part 2

8.1 Question 1: Knapsack Problem

Question: For a Knapsack Problem instance with n items and weight limit W, a neighborhood is defined as follows: Remove the item with the worst value-to-weight ratio from the knapsack and then randomly insert an item that fits into (i.e., such that the weight limit is not exceeded). What is the best upper bound for the size of this neighborhood?

Solution: The best upper bound for the size of this neighborhood is $O(n^2)$.

Explanation: To understand why the upper bound is $O(n^2)$, consider the following:

1. **Value-to-Weight Ratio**: The value-to-weight ratio is a measure of how much value an item provides per unit of weight. Removing the item with the worst ratio means optimizing the knapsack's efficiency.

2. **Neighborhood Size**: After removing one item, you can potentially add any of the remaining $n-1$ items, provided they fit within the weight limit. This gives a linear number of choices.

3. **Combinatorial Nature**: Since you can remove any item and then add any other item, this results in a quadratic number of combinations, hence the $O(n^2)$ complexity.

4. **Generalization**: Even if the question is framed differently, focus on the process of removing and adding items and the combinatorial possibilities to determine the neighborhood size.

8.2 Question 2: Partially Mapped Crossover (PMX)

Question: We apply Partially Mapped Crossover to the following permutations with swapping positions 4 to 7 (in blue):

P1: 9 3 1 7 4 6 2 5 8
P2: 6 4 2 9 8 7 5 3 1

Solution: O1: 6 3 1 9 8 7 5 2 4
O2: 9 8 5 7 4 6 2 3 1

Explanation: To solve PMX problems, follow these steps:

1. **Identify the Crossover Section:** - Positions 4-7 are swapped (blue section) - For O1: Take positions 4-7 from P2 (9875) - For O2: Take positions 4-7 from P1 (7462)

2. **Create Mapping:** - 7 ↔ 9 - 4 ↔ 8 - 6 ↔ 7 - 2 ↔ 5

3. **Fill Remaining Positions:** - For positions outside the crossover section: * If the number from the original parent doesn't create a duplicate, keep it * If it would create a duplicate, use the mapping to find a replacement

8.3 Question 3: Algorithmic Concepts (True/False)

Question: For each statement, determine if it's True, False, or Not sure.

a) TSP instances with up to 50 cities are most easily solved by an Exhaustive Search.
Answer: False
Explanation: While exhaustive search will find the optimal solution, it's not the most efficient approach even for 50 cities. The time complexity would be O(n!), making it impractical. Dynamic programming or branch-and-bound algorithms would be more efficient for this size.

b) If a problem is NP-hard, then it is in NP.
Answer: False
Explanation: This is a common misconception. NP-hard problems are at least as hard as the hardest problems in NP, but they don't necessarily have to be in NP themselves. For example, the halting problem is NP-hard but not in NP.

c) The Steiner Tree Problem is similar to the Minimum Spanning Tree Problem and can hence be solved in polynomial time.
Answer: False
Explanation: While both problems involve connecting nodes in a graph, the Steiner Tree Problem is NP-hard, unlike the Minimum Spanning Tree Problem which can be solved in polynomial time (e.g., using Kruskal's or Prim's algorithm).

d) In the Capacitated Vehicle Routing Problem (CVRP), the location of the depot is usually fixed.
Answer: True
Explanation: In the standard CVRP, the depot location is indeed fixed. The problem focuses on finding optimal routes for vehicles with limited capacity to serve customers from this fixed depot.

e) Tabu Search with a tabu duration t=0 is just like a Local Search.
Answer: True
Explanation: When the tabu duration is 0, no moves are stored in the tabu list, meaning every potential move is immediately "forgotten" and no moves are prohibited.

f) The 3-opt neighborhood for a TSP instance with n cities is of size $O(n^3)$.
Answer: True
Explanation: For 3-opt moves in TSP, we need to select 3 edges from n possible edges. This gives us $\binom{n}{3} = O(n^3)$ possible combinations.

g) The memory used to represent the adjacency matrix of a graph with n vertices and e edges is of size $O(n^2)$.
Answer: True
Explanation: The size is always $n \times n = n^2$ regardless of actual number of edges. Each cell stores 0 or 1 (or weight for weighted graphs).

h) For maximization problems the temperature in Simulated Annealing is increased over time.
Answer: False
Explanation: Temperature always decreases over time, regardless of problem type. High temperature means more random moves accepted.

i) If a problem is NP-complete, then it is proven that there exist no algorithms, which solve the

problem in polynomial time.
Answer: False
Explanation: NP-complete doesn't mean "no polynomial solution exists", it means "no polynomial solution is currently known". If $P \neq NP$ (unproven), then no polynomial solution exists.

j) A Genetic Algorithm with a higher mutation rate usually converges slower but towards better solutions in turn.
Answer: True
Explanation: Higher mutation rates increase exploration of the search space and help escape local optima, but at the cost of slower convergence.

k) Best Insertion is a family of methods for improving existing solutions.
Answer: False
Explanation: It's a construction method, not an improvement method. It builds solutions by inserting elements one at a time.

Key Points to Remember:

- **Algorithm Characteristics:** Understand the fundamental differences between construction methods (like Best Insertion) and improvement methods
- **Complexity Classes:** Know the relationships between P, NP, NP-hard, and NP-complete
- **Space Complexity:** Understand trade-offs between different data structures (e.g., adjacency matrix vs. list)
- **Search Parameters:** Know how parameters affect algorithm behavior (temperature, mutation rate, tabu duration)
- **Problem Variants:** Recognize how problem constraints affect complexity (fixed vs. variable depot in CVRP)

8.4 Question 4: TSP with Asymmetric Distances and Pilot Method

Question: Given is a TSP instance by its asymmetric distance matrix below (e.g. the distance from c3 to c4 equals 7, whereas the distance from c4 to c3 equals 3).

From \To	c1	c2	c3	c4
c1	-	2	9	4
c2	3	-	1	5
c3	3	5	-	7
c4	4	2	3	-

a) (1 point) What is the cost of tour c1-c2-c3-c4-c1?

Solution: Cost = 14

Explanation: To calculate the tour cost:

- c1 → c2: 2
- c2 → c3: 1
- c3 → c4: 7
- c4 → c1: 4
- Total: 2 + 1 + 7 + 4 = 14

Note: In asymmetric TSP, the cost from city i to j may differ from j to i.

b) (5 points) We apply the Pilot Method, and use the Nearest Neighbor heuristic as the pilot strategy. A tour always starts and ends in city c1. What is the cost of the resulting tour?

Solution: Tour: c1-c4-c2-c3-c1
Cost: 10

Explanation: The Pilot Method with Nearest Neighbor works as follows:

1. **Starting Point:** - Start at c1 (required) - Available cities: {c2, c3, c4}

2. **First Step (from c1):** - Try each possible next city and run NN from there - From c1 to c2 (cost=2): Run NN → complete tour cost - From c1 to c3 (cost=9): Run NN → complete tour cost - From c1 to c4 (cost=4): Run NN → gives best complete tour - Choose c4 as it leads to best complete tour

3. **Second Step (from c4):** - Available: {c2, c3} - c4 → c2 (cost=2) is shorter than c4 → c3 (cost=3) - Choose c2

4. **Third Step (from c2):** - Only c3 available - Add c3 (cost=1)

5. **Complete Tour:** - Return to c1 from c3 (cost=3) - Final tour: c1 → c4 → c2 → c3 → c1 - Total cost: 4 + 2 + 1 + 3 = 10

Key Points:

- In asymmetric TSP, always check the correct direction costs
- Pilot Method looks ahead using a simple heuristic (NN here)
- The method makes each choice based on complete tour costs
- This often gives better results than simple Nearest Neighbor alone

8.5 Question 5: Local Search and Tabu Search

Question: (15 points) The values of a function f(x, y) of two integer variables defined on the domain [-7, 7] × [-7, 7] are given by a table. We define a neighborhood that allows adding or subtracting 1 to/from one of the two variables (i.e. horizontal or vertical moves, but no diagonal ones).

a) (5 points) The **minimum** of f is determined using a **Local Search** parametrized as follows:

- Initialization: x = -7, y = -7, f(x, y) = 175
- Selection Criterion: First Improving Move
- Ordering of the moves: Right, Up, Left, Down

Solution: 175-153-125-104-88-60-39-24-15-11-24-49-59-81-109-stop

Key Points for Local Search:

- Always check neighbors in specified order (Right, Up, Left, Down)
- Take first move that improves (reduces) the value
- Stop when no improving move exists
- Draw arrows on table to track moves during exam

b) (10 points) The **minimum** of f is determined using a **Tabu Search** parametrized as follows:

- Initialization: x = -7, y = -7, f(x, y) = 175
- Tabu Condition: If t is added to a variable, there is no subtraction from this variable allowed for t iterations. Same if t is subtracted.
- The Tabu Condition applies unless a step leads to an improvement of the best solution so far.
- Selection Criterion: Always take the best possible move
- Stopping criteria: Maximum of 14 iterations reached, or no more moves allowed

Solution for t = 4: 175-133-85-46-18-13-14-10-10-9-9-7-13-10-3-stop

Key Points for Tabu Search:

- Keep track of tabu moves for each variable (x and y)
- If you add to x, can't subtract from x for t iterations
- If you subtract from x, can't add to x for t iterations
- Same rules apply to y
- Aspiration: Take move anyway if it leads to best solution so far
- Always take best available non-tabu move

Exam Strategy:

- Draw arrows on the table to track your moves
- Keep a tabu list for each variable (x and y)
- For each step write:
  - Current position (x,y)
  - Current value f(x,y)
  - Available moves
  - Tabu status
  - Best move chosen

8.6 Question 6: Santa's Sleigh Challenge

Question: (3 points) We consider the Santa's Sleigh Challenge. Please answer ONE of the following two questions with at most 500 characters:

a) Briefly describe a strategy to find an initial solution. **OR**

b) Briefly describe a strategy to improve an existing solution.

Solution Approach for Initial Solution:

- Use Nearest Neighbor heuristic: Start from North Pole, always visit closest gift next
- Consider weight constraints when building routes
- Split into multiple trips when sleigh capacity is reached
- Balance between distance and weight capacity

Solution Approach for Improvement:

- Apply 2-opt or 3-opt moves within each route
- Try moving gifts between adjacent routes
- Merge short routes and split long ones
- Use simulated annealing to escape local optima
- Focus on heaviest gifts first as they have most impact

Key Points:

- Consider both distance and weight constraints
- Balance between route length and number of trips
- Local search moves should respect weight capacity
- Prioritize improvements on longest/heaviest routes

8.7 Question 7: TSP Lower Bound

Question: (3 points) Consider the Traveling Salesperson Problem instance sw24978. Give a good lower bound estimate of the number of possible tours, as a power of 10.

Solution: Number of cities: 24978
Number of tours $> 10^{51161}$

Explanation: For a TSP with n cities:

- Total possible tours = (n-1)!/2
- For n = 24978: $\log_{10}((24978-1)!/2) > 51161$
- This shows the enormous size of the solution space

Key Points for Similar Questions:

- **When analyzing algorithm complexity:**
  - Check if problem is in P, NP, or NP-hard
  - Look for special cases that might be easier
  - Consider if approximation algorithms exist
  - Understand the difference between decision and optimization problems
- **When evaluating metaheuristics:**
  - Consider how parameters affect performance
  - Look for problem-specific adaptations
  - Understand exploration vs exploitation trade-off
  - Know typical parameter ranges and their effects
- **For selection methods in evolutionary algorithms:**
  - Roulette wheel: probability proportional to fitness
  - Tournament: select best from random subset
  - Rank-based: probability based on sorted position
  - Consider selection pressure and diversity
- **For neighborhood structures:**
  - Must be connected (can reach all solutions)
  - Size affects computational effort
  - Should reflect problem structure
  - Consider move evaluation efficiency

8.8 Question 8: 2-opt Move

Question: (3 points) For a Travelling Salesperson Problem instance with seven cities, consider the tour: c_1-c_2-c_3-c_4-c_5-c_6-c_7-c_1

Give the resulting tour of applying the 2-opt move, where the edges c_2-c_3 and c_5-c_6 are replaced.

Solution: Tour: c_1-c_2-c_5-c_4-c_3-c_6-c_7-c_1

Key Points for 2-opt:

- Remove two edges
- Reverse the path between them
- Reconnect with new edges
- Useful for removing path crossings

Key Points for Similar Questions:

- **For routing problems:**
  - Check all constraints (capacity, time windows, etc.)
  - Consider vehicle characteristics
  - Understand cost calculation rules
  - Look for problem-specific features
- **When analyzing distance metrics:**
  - Euclidean vs Manhattan vs Great Circle
  - Symmetric vs asymmetric distances
  - Consider if triangle inequality holds
  - Impact on solution methods
- **For multi-trip problems:**
  - Consider depot location and return trips
  - Balance load across trips
  - Account for vehicle capacity
  - Look for time/distance constraints
- **When evaluating solution methods:**
  - Construction vs improvement methods
  - Single vs multiple vehicle approaches
  - Local vs global optimization
  - Trade-off between quality and speed

8.9 Question 9: CVRP Neighborhood

Question: (7 points) Consider an instance of the Capacitated Vehicle Routing Problem (CVRP) with n customers and vehicle capacity Q. We define a neighborhood of a given solution S with m non-empty tours as follows:

- Select a random tour t_s (source tour)
- Select a random customer X from t_s
- Traverse all other tours t_d in a random ordering:
  - Find a customer Y in t_d such that the tours of X and Y can be swapped without exceeding the vehicle capacity Q for both tours

- If such Y is found, then insert X in t_d and Y in t_s at random positions and then EXIT the traversing loop

a) (4 points) Give a good estimate of the running time (in O-Notation) to compute a new solution from this neighborhood. Hereby, assume that computing the total weight of a tour in the current solution takes O(1).

Solution: Running time: O(n)

Explanation:

- Selecting random tour and customer: O(1)
- For each other tour (max m-1 tours):
  - Check each customer Y in current tour: O(n/m)
  - Weight calculation: O(1)
  - Total per tour: O(n/m)
- Total complexity: O(m * n/m) = O(n)

Key Points for Similar Questions:

- **For ant colony optimization:**
  - Understand pheromone update rules
  - Know how solutions are constructed
  - Consider parameter settings:
    - Colony size vs problem size
    - Evaporation rate effects
    - Pheromone bounds
    - Local vs global updates
  - Recognize convergence behavior
- **For population-based methods:**
  - Population size considerations
  - Diversity maintenance
  - Selection pressure effects
  - Convergence criteria
- **When analyzing cooperative algorithms:**
  - Information sharing mechanisms
  - Synchronization points
  - Local vs shared memory
  - Solution combination strategies
- **For parameter adaptation:**
  - Static vs dynamic parameters
  - Feedback mechanisms
  - Problem size scaling
  - Performance indicators

b) (3 points) Give a good upper bound estimate for the size of the neighborhood of S.

Solution: Size: $O(n^4)$

Explanation:

- Number of source tours: m
- Customers per tour: O(n/m)
- Possible destination tours: m-1
- Possible positions in destination tour: O(n/m)
- Possible positions in source tour: O(n/m)
- Total: O(m * n/m * m * n/m * n/m) = $O(n^4)$

Key Points for Similar Questions:

- **For routing problems:**
  - Check all constraints (capacity, time windows, etc.)
  - Consider vehicle characteristics
  - Understand cost calculation rules
  - Look for problem-specific features
- **When analyzing distance metrics:**
  - Euclidean vs Manhattan vs Great Circle
  - Symmetric vs asymmetric distances
  - Consider if triangle inequality holds
  - Impact on solution methods
- **For multi-trip problems:**
  - Consider depot location and return trips
  - Balance load across trips
  - Account for vehicle capacity
  - Look for time/distance constraints
- **When evaluating solution methods:**
  - Construction vs improvement methods
  - Single vs multiple vehicle approaches
  - Local vs global optimization
  - Trade-off between quality and speed

8.10 Question 10: True/False Statements (5.5 points)

Note: For every correct answer you will get 0.5 point, for every incorrect answer 0.5 points will be subtracted. For every "Not sure" answer you will neither get nor lose any points. Maximum amount of points for this task is 5.5 points, minimum is 0 points, i.e. a negative total will be set to 0.

a) Vertex Coloring for graphs in general is NP-hard.

- ▶ True (correct)
- ○ False
- ○ Not sure

Key Points for Similar Questions:

- NP-hard problems typically involve:
  - Finding optimal solutions in large search spaces
  - No known polynomial-time algorithms

  - Often have special cases that are polynomial-time solvable
- For graph coloring specifically:
  - 2-coloring (bipartite) is polynomial
  - 3+ coloring is NP-hard
  - Greedy coloring gives upper bound

b) Although the Steiner Tree Problem is theoretically interesting, it is of no great practical use.

- ○ True
- ▶ False (correct)
- ○ Not sure

Key Points for Similar Questions:

- Real-world applications to consider:
  - Network design (telecommunications, power grids)
  - Circuit design in VLSI
  - Transportation networks
  - Pipeline systems
- When evaluating practical use:
  - Look for real industry applications
  - Consider if problem appears as subproblem
  - Check if approximation algorithms exist

c) Genetic Algorithms are constructive methods.

- ○ True
- ▶ False (correct)
- ○ Not sure

Key Points for Similar Questions:

- Understanding algorithm types:
  - Constructive: Build solution step by step (e.g., Greedy)
  - Improvement: Start with complete solution and modify (e.g., Local Search)
  - Population-based: Work with multiple solutions (e.g., Genetic Algorithms)
- Genetic Algorithm characteristics:
  - Requires initial population
  - Modifies existing solutions
  - Uses selection, crossover, mutation

d) Genetic Algorithms are suitable for solving Knapsack Problems, since Crossover and Mutation are easy to define for bit vectors.

- ▶ True (correct)
- ○ False
- ○ Not sure

Key Points for Similar Questions:

- When is a problem suitable for Genetic Algorithms:
  - Natural binary representation possible
  - Easy to define crossover/mutation
  - Solution space is large
  - Multiple local optima likely
- For binary problems specifically:
  - Bit-flip mutation is straightforward
  - One-point/two-point crossover works well
  - Easy fitness function calculation

e) In the Roulette Wheel Method solutions are chosen with uniformly distributed probability.

- ○ True
- ▶ False (correct)
- ○ Not sure

Key Points for Similar Questions:

- Selection methods to consider:
  - Roulette Wheel: Probability proportional to fitness
  - Random Sampling: Uniform probability
  - Tournament: Compare subset of solutions
- When evaluating selection methods:
  - Consider bias towards better solutions
  - Look for computational efficiency
  - Check if method promotes diversity

f) Minimum Shortest Path Problem is in NP.

- ▶ True (correct)
- ○ False
- ○ Not sure

Key Points for Similar Questions:

- Complexity classes to remember:
  - P: Polynomial time solvable
  - NP: Verifiable in polynomial time
  - NP-hard: At least as hard as hardest NP problems
  - NP-complete: NP-hard and in NP
- For shortest path problems:
  - Dijkstra's algorithm solves in polynomial time
  - Negative weight edges require Bellman-Ford
  - NP-hard variants exist (e.g., with constraints)

g) Choosing an appropriate neighborhood is crucial for solving a problem with Simulated Annealing.

- ▶ True (correct)
- ○ False
- ○ Not sure

Key Points for Similar Questions:

- Neighborhood design considerations:
  - Connectivity: Ensure all solutions reachable
  - Size: Balance exploration and computational cost
  - Structure: Reflect problem's underlying structure
- When evaluating neighborhood quality:
  - Check if it allows escaping local optima
  - Consider the impact on convergence speed
  - Look for problem-specific neighborhood designs

h) Random Sampling converges fast towards good solutions if the parameters of the algorithm are set appropriately.

- ○ True
- ▶ False (correct)
- ○ Not sure

Key Points for Similar Questions:

- Random Sampling characteristics:
  - Simple to implement
  - Fast convergence unlikely
  - May get stuck in local optima
- When evaluating convergence speed:
  - Consider the size of the solution space
  - Look for problem-specific heuristics
  - Check if algorithm uses learning/adaptation

i) In Simulated Annealing, the temperature schedule must be chosen depending on the size of the problem to be solved.

- ▶ True (correct)
- ○ False
- ○ Not sure

Key Points for Similar Questions:

- Temperature schedule considerations:
  - Initial temperature: High enough for exploration
  - Cooling rate: Balance exploration and convergence
  - Final temperature: Low enough for convergence
- When evaluating temperature schedules:
  - Check if it allows escaping local optima
  - Consider the impact on convergence speed
  - Look for problem-specific temperature schedules

j) Capacitated Vehicle Routing Problems (CVRP) can always be solved optimally with a greedy algorithm.

- ○ True
- ▶ False (correct)
- ○ Not sure

Key Points for Similar Questions:

- CVRP complexity:
  - NP-hard due to routing and capacity constraints
  - Greedy algorithms give fast but not optimal solutions
  - Exact methods exist but are computationally expensive
- When evaluating solution methods:
  - Consider the trade-off between solution quality and computational time
  - Look for problem-specific heuristics
  - Check if algorithm uses learning/adaptation

k) Tabu Search is an improving method.

- ▶ True (correct)
- ○ False
- ○ Not sure

Key Points for Similar Questions:

- Tabu Search characteristics:
  - Uses memory to store recently visited solutions
  - Avoids cycling by forbidding recent moves
  - Can escape local optima using aspiration criteria
- When evaluating improving methods:
  - Consider the trade-off between solution quality and computational time
  - Look for problem-specific heuristics
  - Check if algorithm uses learning/adaptation

Key Points to Remember:

- **Algorithm Characteristics:** Understand the fundamental differences between construction methods (like Best Insertion) and improvement methods
- **Complexity Classes:** Know the relationships between P, NP, NP-hard, and NP-complete
- **Space Complexity:** Understand trade-offs between different data structures (e.g., adjacency matrix vs. list)
- **Search Parameters:** Know how parameters affect algorithm behavior (temperature, mutation rate, tabu duration)
- **Problem Variants:** Recognize how problem constraints affect complexity (fixed vs. variable depot in CVRP)

8.11 Question 11: Santa's Sleigh Challenge (3 points)

Note: For every correct answer you will get 0.5 point, for every incorrect answer 0.5 points will be subtracted. For every "Not sure" answer you will neither get nor lose any points. Maximum amount of points for this task is 5.5 points, minimum is 0 points, i.e. a negative total will be set to 0.

We consider the Santa's Sleigh Challenge. Which of the following statements is correct?

a) Distances between locations are computed using Euclidean coordinates.

- ○ True
- ▶ False (correct)
- ○ Not sure

Key Points for Similar Questions:

- When analyzing distance calculations:
  - Check if Euclidean, Manhattan, or Great Circle
  - Consider if distances are symmetric
  - Look for special cases (e.g., grid-based)
- For routing problems:
  - Check capacity constraints
  - Consider time windows if present
  - Look for multiple vehicle aspects
  - Identify if split deliveries allowed

b) The goal of the challenge is to minimize the total amount of time used to deliver all the presents.

- ○ True
- ▶ False (correct)
- ○ Not sure

Key Points for Similar Questions:

- Objective functions to consider:
  - Minimize total distance
  - Minimize total time
  - Maximize profit
  - Minimize environmental impact
- When evaluating objective functions:
  - Consider the problem's context
  - Look for multiple objectives
  - Check if objectives are conflicting

c) Presents may be temporarily stored at any location.

- ○ True
- ▶ False (correct)
- ○ Not sure

Key Points for Similar Questions:

- Problem constraints to consider:
  - Capacity constraints
  - Time windows
  - Multiple vehicles
  - Split deliveries
- When evaluating problem constraints:
  - Check if constraints are hard or soft
  - Consider the impact on solution quality
  - Look for problem-specific constraints

d) The cost of a tour depends on the weight of the sleigh.

- ▶ True (correct)
- ○ False
- ○ Not sure

Key Points for Similar Questions:

- Cost functions to consider:
  - Distance-based
  - Time-based
  - Weight-based
  - Profit-based
- When evaluating cost functions:
  - Consider the problem's context
  - Look for multiple cost components
  - Check if costs are linear or non-linear

e) The weight of the sleigh is higher than its carrying capacity.

- ○ True
- ▶ False (correct)
- ○ Not sure

Key Points for Similar Questions:

- Vehicle characteristics to consider:
  - Capacity
  - Weight
  - Speed
  - Fuel efficiency
- When evaluating vehicle characteristics:
  - Check if characteristics are fixed or variable
  - Consider the impact on solution quality
  - Look for problem-specific characteristics

f) The winning team of the official challenge used an exclusively greedy method to find their final solution.

- ○ True
- ▶ False (correct)
- ○ Not sure

Key Points for Similar Questions:

- Solution methods to consider:
  - Greedy algorithms
  - Local search
  - Metaheuristics
  - Exact methods
- When evaluating solution methods:
  - Consider the trade-off between solution quality and computational time
  - Look for problem-specific heuristics
  - Check if algorithm uses learning/adaptation

Key Points to Remember:

- **Problem Context:** Understand the problem's context and constraints
- **Objective Functions:** Know the different types of objective functions
- **Cost Functions:** Understand the different types of cost functions
- **Vehicle Characteristics:** Know the different vehicle characteristics
- **Solution Methods:** Understand the different solution methods

8.12 Question 12: Ant System Algorithm (2 points)

Note: For every correct answer you will get 0.5 point, for every incorrect answer 0.5 points will be subtracted. For every "Not sure" answer you will neither get nor lose any points. Maximum amount of points for this task is 5.5 points, minimum is 0 points, i.e. a negative total will be set to 0.

A basic Ant System algorithm (i.e. without any local search application to the intermediate solutions) is applied to the Traveling Sales Person problem (TSP). Which of the following statements are correct?

a) Choosing an appropriate neighborhood is crucial for solving the problem with the Ant System algorithm.

- ○ True
- ▶ False (correct)
- ○ Not sure

Key Points for Similar Questions:

- Algorithm characteristics to check:
  - Population vs single solution
  - Memory usage (e.g., pheromone trails)
  - Parallel vs sequential
  - Deterministic vs probabilistic
- Implementation aspects:
  - Parameter settings (colony size, evaporation rate)
  - Solution construction method
  - Update rules
  - Stopping criteria

b) When building a path, ants choose an edge with a probability proportional to the amount of pheromones accumulated on that edge.

- ▶ True (correct)
- ○ False
- ○ Not sure

Key Points for Similar Questions:

- Pheromone trails:
  - Used to guide search
  - Updated based on solution quality
  - Evaporate over time
- When evaluating pheromone trails:
  - Check if they promote exploration
  - Consider the impact on convergence speed
  - Look for problem-specific pheromone trail designs

c) Inside every iteration of the algorithm, artificial ants cooperate in building a new path.

- ○ True
- ▶ False (correct)
- ○ Not sure

Key Points for Similar Questions:

- Cooperation mechanisms:
  - Pheromone trails
  - Solution sharing
  - Cooperative problem solving
- When evaluating cooperation mechanisms:
  - Check if they promote exploration
  - Consider the impact on convergence speed
  - Look for problem-specific cooperation mechanisms

d) To get good solutions, the number of ants must be chosen depending on the number of cities of the TSP instance to be solved.

- ▶ True (correct)
- ○ False
- ○ Not sure

Key Points for Similar Questions:

- Parameter settings:
  - Colony size
  - Evaporation rate
  - Solution construction method
  - Update rules
- When evaluating parameter settings:
  - Check if they promote exploration
  - Consider the impact on convergence speed
  - Look for problem-specific parameter settings

Key Points to Remember:

- **Algorithm Characteristics:** Understand the fundamental differences between population-based and single-solution algorithms
- **Pheromone Trails:** Know how pheromone trails are used to guide search
- **Cooperation Mechanisms:** Understand how cooperation mechanisms promote exploration and convergence
- **Parameter Settings:** Know how parameter settings affect algorithm performance
- **Problem-Specific Designs:** Look for problem-specific designs for pheromone trails, cooperation mechanisms, and parameter settings

General Tips for Multiple Choice:

- For algorithm questions: Consider if it's constructive or improving
- For complexity questions: Think about whether problem can be solved in polynomial time
- For parameter questions: Consider if parameter affects solution quality
- For probability questions: Think about whether it's uniform or weighted
- When unsure: "Not sure" is better than guessing (no point deduction)